CRAY
RESEARCH, INC.

CRAY COMPUTER SYSTEMS

CRAY X-MP SERIES
MODEL 48
MAINFRAME REFERENCE MANUAL

HR-0097

RECORD OF REVISION CRAY RESEARCH, INC. PUBLICATION NUMBER HR-0097

Revision	Description
	August, 1984 - Original printing.

PREFACE

This publication describes the CRAY X-MP Series Model 48 Computer System. It is written to assist programmers and engineers and assumes a familiarity with digital computers.

The manual describes the overall computer system, its configurations, and equipment. It also describes the operation of the Central Processing Units that execute instructions, provide memory protection, report hardware exceptions, and provide interprocessor communications within the system.

Details of the I/O Subsystem, the disk storage units, and the Solid-state Storage Device are given in the following publications:

HR-0030	I/O Subsystem Hardware Reference Manual
HR-0630	Mass Storage Subsystem Hardware Reference Manual
HR-0031	Solid-state Storage Device (SSD®) Reference Manual

//

WARNING

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

//

CONTENTS

FIGURES

FIGURES (continued)

TABLES

SYSTEM DESCRIPTION 1

INTRODUCTION

The CRAY X-MP model 48 Computer System is a powerful, general purpose machine that contains four central processing units (CPUs). Like all CRAY X-MP multiprocessor systems, it is able to achieve extremely high multiprocessing rates by efficiently using the scalar and vector capabilities of all CPUs combined with the system's random-access solid-state memory (RAM) and shared registers.

Vector processing is the performance of iterative operations on sets of ordered data. When two or more vector operations are chained together, two or more operations can be executing each 9.5-nanosecond clock period, greatly exceeding the computational rates of conventional scalar processing. Scalar operations complement the vector capability by providing solutions to problems not readily adaptable to vector techniques.

The machine has very high performance levels, and equipment options allow systems to be configured for a particular use. Central Memory of the 4-processor mainframe is 8 million 64-bit words (see table 1-1). The system is compatible with all existing models of the Cray I/O Subsystem and its associated mass storage subsystem. In addition, an optional high-performance Cray Solid-state Storage Device (SSD) can be attached to the mainframe. Figure 1-1 illustrates the mainframe with a Cray I/O Subsystem and an SSD.

This section describes system components and configurations. Table 1-1 gives overall system characteristics.

CONVENTIONS

The following conventions are used in this manual.

ITALICS

Italicized lowercase letters, such as *jk*, indicate variable information.

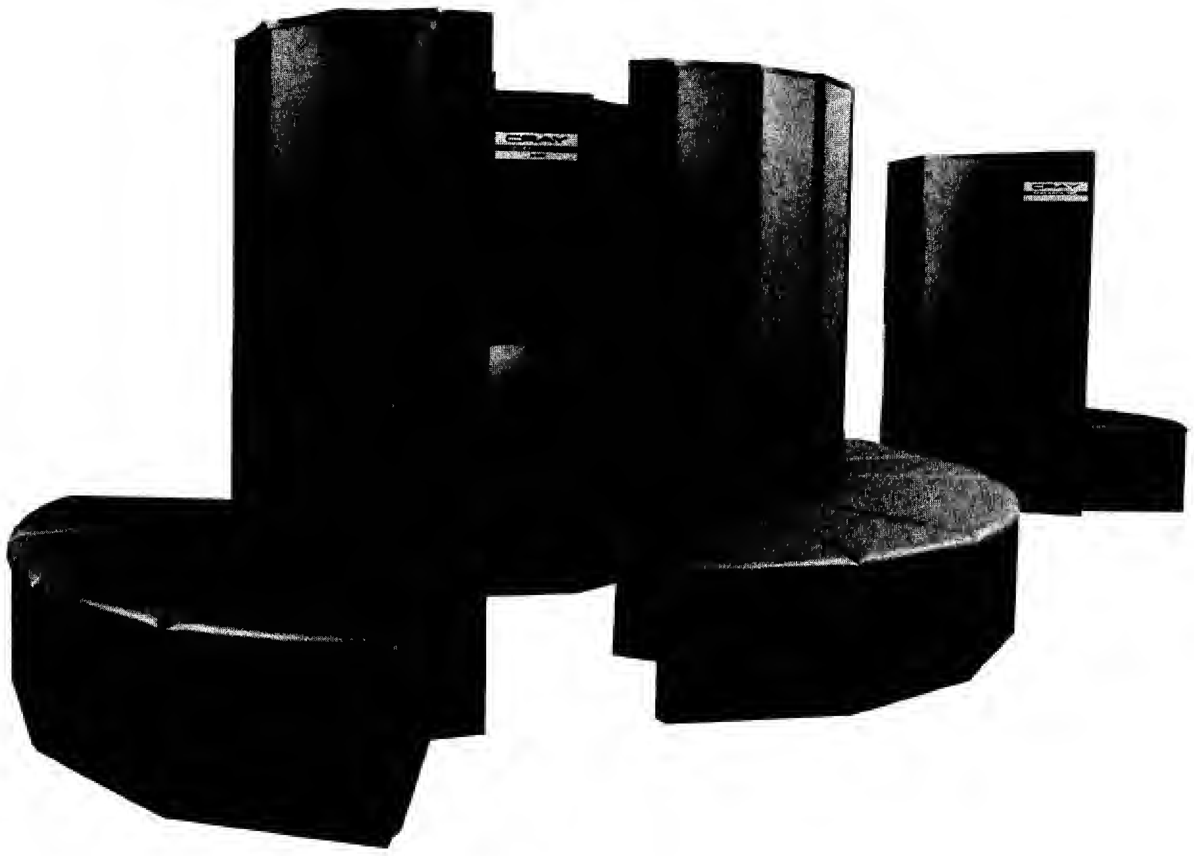


Figure 1-1. CRAY X-MP model 48 mainframe with a Cray I/O Subsystem and an SSD

Table 1-1. CRAY X-MP 4-processor system characteristics

Configuration	- Mainframe with 4 Central Processing Units (CPUs) - I/O Subsystem with 2, 3, or 4 I/O Processors - Optional Solid-state Storage Device (SSD)
CPU speed	- 9.5 ns CPU clock period - 105 million floating-point additions per second per CPU - 105 million floating-point multiplications per second per CPU - 105 million half-precision floating-point divisions per second per CPU - 33 million full-precision floating-point divisions per second per CPU - Simultaneous floating-point addition, multiplication, and reciprocal approximation within each CPU
Memory	- Mainframe has 8 million (model 48) 64-bit words in Central Memory
Input/Output	- Two 1250 Mbyte per second channel pairs for interface to Solid-state Storage Device (SSD) - Four 100 Mbyte per second channel pairs for interface to I/O Subsystem - Four 6 Mbyte per second channel pairs
Physical	- 64 sq ft floor space for mainframe - 15 sq ft floor space for I/O Subsystem - 15 sq ft floor space for SSD - 5.65 tons, mainframe weight - 1.5 tons, I/O Subsystem weight - 1.5 tons, SSD weight - Liquid refrigeration of each chassis - 400 Hz power from motor-generators

REGISTER CONVENTIONS

Parenthesized register names are used frequently in this manual as a form of shorthand notation for the expression "the contents of register ---." For example, "Branch to (P)" means "Branch to the address indicated by the contents of register P."

Designations for the A, B, S, T, and V registers are used extensively. For example, "Transmit (T*jk*) to S*i*" means "Transmit the contents of the T register specified by the *jk* designators to the S register specified by the *i* designator."

Register bits are numbered right to left as powers of 2, starting with 2^0. Bit 2^{63} of an S, V, or T register value represents the most significant bit. Bit 2^{23} of an A or B register value represents the most significant bit. (A and B registers are 24 bits.) The numbering conventions for the Exchange Package and the Vector Mask register are exceptions. Bits in the Exchange Package are numbered from left to right and are not numbered as powers of 2 but as bits 0 through 63 with 0 as the most significant and 63 as the least significant. The Vector Mask register has 64 bits, each corresponding to a word element in a vector register. Bit 2^{63} corresponds to element 0, bit 2^0 corresponds to element 63.

NUMBER CONVENTIONS

Unless otherwise indicated, numbers in this manual are decimal numbers. Octal numbers are indicated with an 8 subscript. Exceptions are register numbers, channel numbers, instruction parcels in instruction buffers, and instruction forms which are given in octal without the subscript.

CLOCK PERIOD

The basic unit of CPU computation time is 9.5 nanoseconds (ns) and is referred to as a clock period (CP). Instruction issue, memory references, and other timing considerations are often measured in CPs.

SYSTEM COMPONENTS

The 4-processor system is composed of a mainframe and an I/O Subsystem. Mass storage devices, front-end interfaces, and optional tape devices are also integral parts of a system. Optionally, a Cray Solid-state Storage Device (SSD) can be part of the system. Supporting this equipment are condensing units for refrigeration, motor-generators to provide system power, and power distribution units for the mainframe, I/O Subsystem, and SSD. System components are described on the following pages.

CENTRAL PROCESSING UNITS

Each CPU has independent control and computation sections. All CPUs share Central Memory and the inter-CPU communication and I/O sections. (CPU sections are described in later sections.) Figure 1-2 shows the mainframe chassis. Figure 1-2 illustrates the basic organization of the computer; figure 1-3 illustrates the components and control and data paths of each CPU in the system.

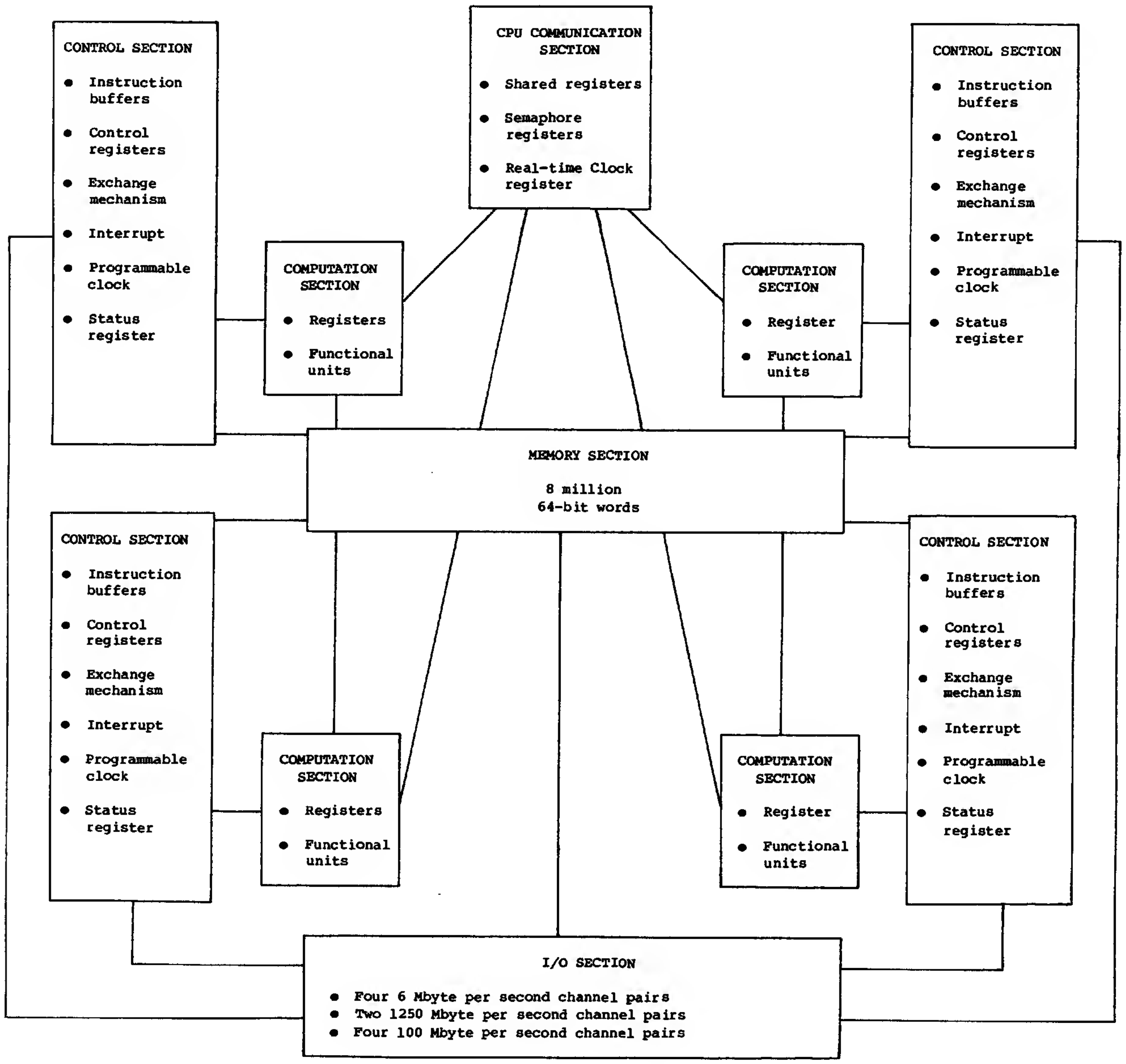
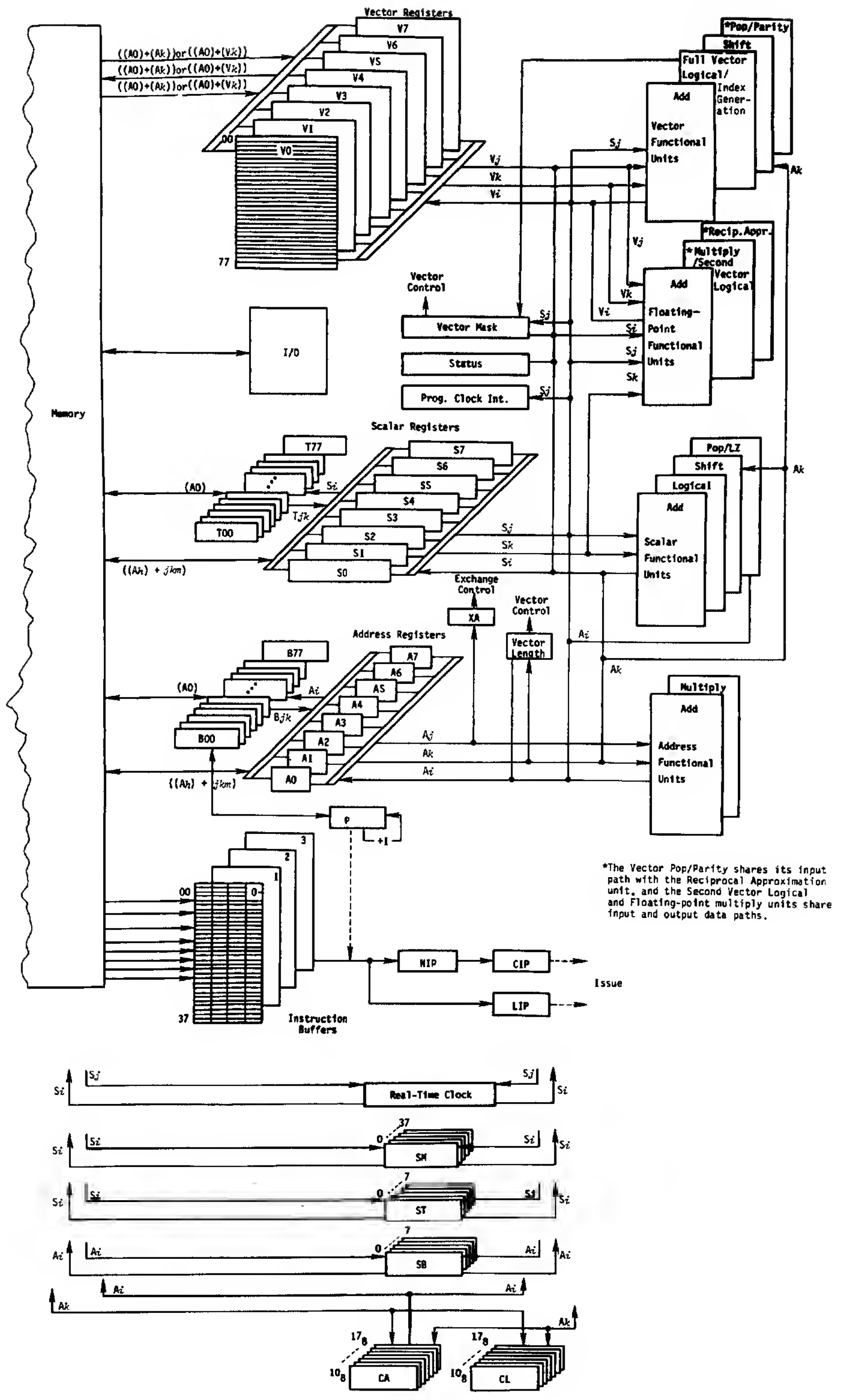


Figure 1-2. Basic organization of the 4-processor system



*The Vector Pop/Parity shares its input path with the Reciprocal Approximation unit, and the Second Vector Logical and Floating-point multiply units share input and output data paths.

Figure 1-3. Control and data paths for a single CPU

INTERFACES

The Cray system is designed for use with front-end computers in a computer network. A front-end computer system is self contained and executes under the control of its own operating system.

Standard interfaces connect the Cray mainframe's I/O channels to channels of front-end computers, providing input data to the Cray system and receiving output from it for distribution to peripheral equipment. Interfaces compensate for differences in channel widths, machine word size, electrical logic levels, and control signals. (The Master I/O Processor of the I/O Subsystem communicates with the mainframe through a 6 Mbyte per second channel pair to a channel adapter module in the Cray mainframe.) Communication continues through a front-end interface, to the front-end computer typically through a front-end computer I/O channel.

The front-end interface is housed in a stand-alone cabinet (figure 1-4) located near the host computer. Its operation is invisible to the front-end computer user and the Cray user.

A primary goal of the interface is to maximize the use of the front-end channel connected to the Cray system. Since the MIOP channel connected to the interface is faster than any front-end channel connected to the interface, the burst rate of the interface is limited by the maximum rate of the front-end channel.

Interfaces to front-end computers allow the front-end computers to service the Cray Computer System in the following ways:

- As a master operator station
- As a local operator station
- As a local batch entry station
- As a data concentrator for multiplexing several other stations into a single Cray channel
- As a remote batch entry station
- As an interactive communication station

Peripheral equipment attached to the front-end computer varies depending on the use of the Cray system.

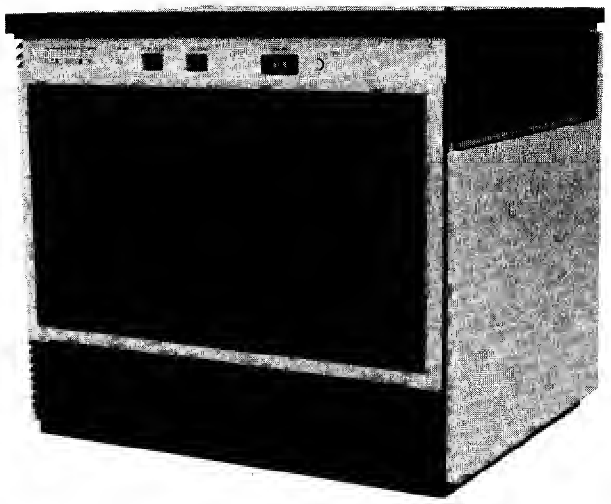


Figure 1-4. Typical interface cabinet

I/O SUBSYSTEM

The I/O Subsystem, shown in figure 1-5, is standard on the CRAY X-MP system and has two, three, or four I/O Processors (IOPs), Buffer Memory, and required interfaces. The I/O Subsystem is designed to provide fast data transfer between its Buffer Memory and the mainframe's Central Memory as well as front-end computers, peripheral devices, and storage devices.

Four types of I/O Processors may be configured in an I/O Subsystem: a Master IOP (MIOP), a Buffer IOP (BIOP), a Disk IOP (DIOP) and an Auxiliary IOP (XIOP). All I/O Subsystems must have at least one MIOP and one BIOP. The number of DIOPs and XIOPs is site dependent.

Each IOP of the I/O Subsystem has a memory section, a control section, a computation section, and an input/output section. Input/output sections are independent and handle some portion of the I/O requirements for the Subsystem. Each IOP also has six direct memory access ports to its Local Memory.

The Master I/O Processor (MIOP) controls the front-end interfaces and the standard group of station[†] peripherals. The Peripheral Expander interfaces the station peripherals to one direct memory access (DMA) port of the MIOP. The MIOP also connects to Buffer Memory and to the

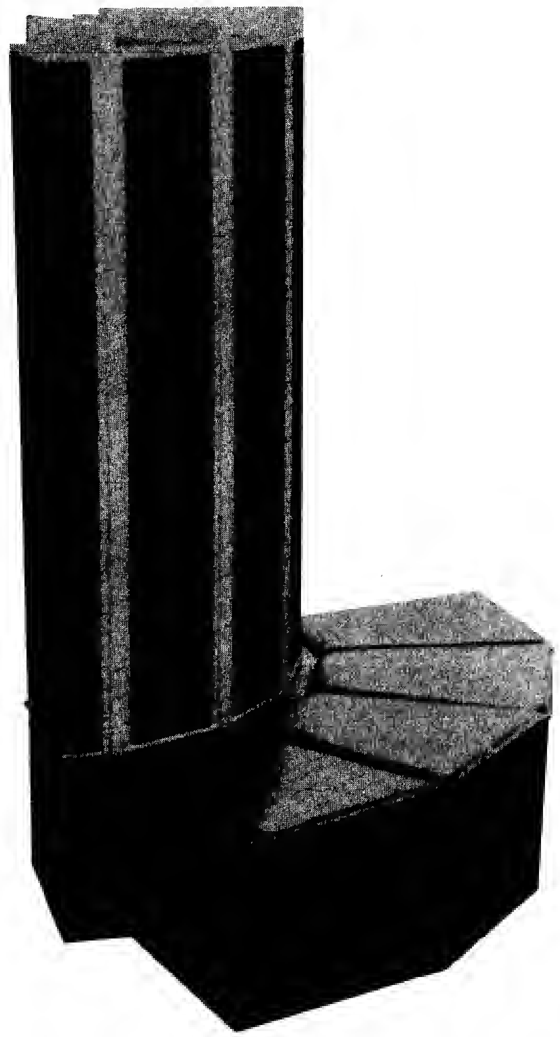


Figure 1-5. I/O Subsystem chassis

† The term station means both hardware and software. Station is the link to the front end or can act as a limited front end (as the MIOP).

mainframe over a 6 Mbyte per second channel pair. The MIOP communicates with the Cray Operating System (COS) to coordinate the activities of the entire I/O Subsystem.

The Buffer I/O Processor (BIOP) is the main link between the mainframe's Central Memory and the mass storage devices. Data from mass storage is transferred through the BIOP's Local Memory to the mainframe's Central Memory through a 100 Mbyte per second channel pair.

The Disk I/O Processor (DIOP) is used for additional disk storage units. This processor can handle up to four disk controller units with up to 16 disk storage units. The DIOP uses one DMA port for each controller, one DMA port to connect to Buffer Memory, and another DMA port to connect a 100 Mbyte per second channel pair to the mainframe Central Memory.

The Auxiliary I/O Processor (XIOP) is used for block multiplexer channels and interfaces to a maximum of four BMC-4 Block Multiplexer Controllers. Each controller can handle up to four block multiplexer channels. The XIOP uses one DMA port for each controller and another DMA port to connect with Buffer Memory.

I/O Subsystem hardware allows for simultaneous data transfers between the BIOP and DIOP or XIOP of the I/O Subsystem and the mainframe's Central Memory.†

The CPU input/output section for the system is described in section 2 of this manual. Refer to the I/O Subsystem Reference Manual, CRI publication HR-0030, for a complete description of the I/O Subsystem.

DISK STORAGE UNITS

For mass storage, the system uses Cray Research, Inc., disk storage units (DSUs). A disk controller unit (DCU) interfaces the disk storage units with an I/O Processor of an I/O Subsystem through one direct memory access (DMA) port. Up to four disk storage units can be connected to a single DCU.

The I/O Processor and the disk controller unit can transfer data between the DMA port and four DSUs with all DSUs operating at full speed without missing data or skipping revolutions. A minimum of 2 and a maximum of 48 DSUs can be configured on an I/O Subsystem. Figure 1-6 shows a Cray DD-49 Disk Storage Unit. The disk controller unit (DCU) is housed in the I/O Subsystem chassis.

† Software to support the 100 Mbyte per second channel pair to the XIOP is currently not available.

Each DSU has two accesses for connecting it to controllers. The second independent data path to each DSU exists through another Cray Research, Inc., controller. Reservation logic provides controlled access to each DSU. Dynamic sharing of devices is not supported by the Cray Operating System (COS) software. Further information about the mass storage subsystem is included in the I/O Subsystem Reference Manual, CRI publication HR-0030, and the Mass Storage Subsystem Hardware Reference Manual, CRI publication HR-0630.

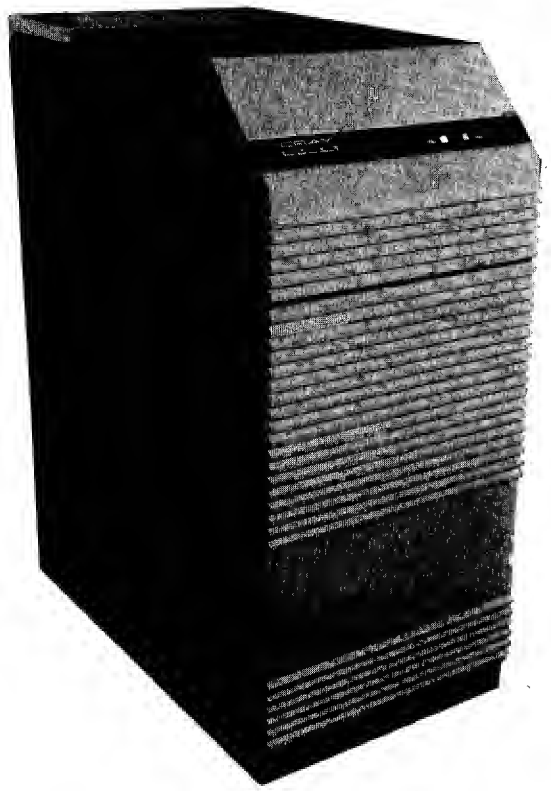


Figure 1-6. DD-49 Disk Storage Unit

SOLID-STATE STORAGE DEVICE

The Solid-state Storage Device (SSD) shown in figure 1-7 is used for temporary data storage and transfers data to and from the mainframe's Central Memory. The transfer speed is dependent on the SSD memory size and configuration as described in the Solid-state Storage Device (SSD) Reference Manual, CRI publication HR-0031. The maximum speed attained from the SSD to Central Memory is 1250 Mbytes per second for each 1250 Mbyte channel.

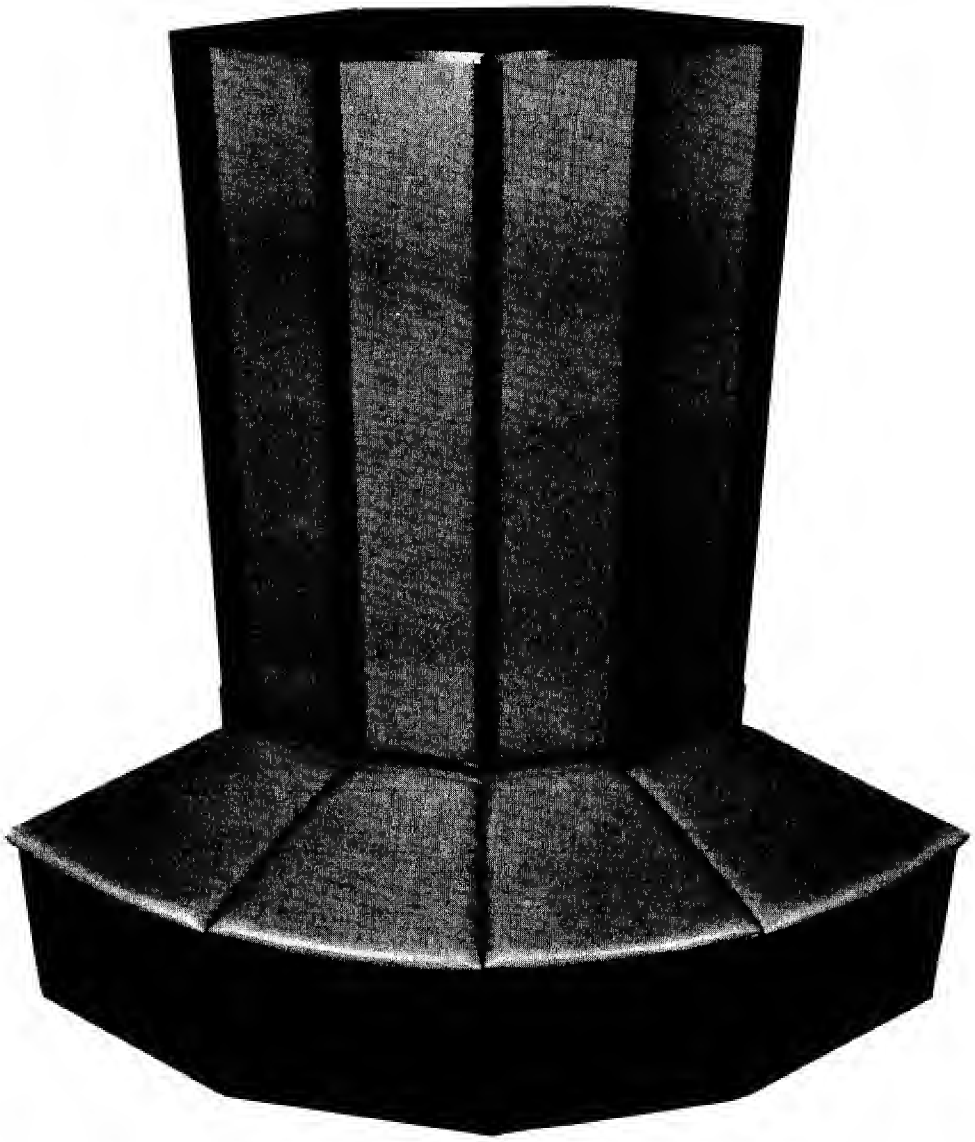


Figure 1-7. Solid-state Storage Device chassis

CONDENSING UNITS

Condensing units (figure 1-8) contain the major components of the refrigeration system used to cool the computer chassis and consist of two 25-ton condensers. Heat is removed from the condensing unit by a second level cooling system that is not part of the computer system. Freon, which cools the computer, picks up heat and transfers it to water in the condensing unit.

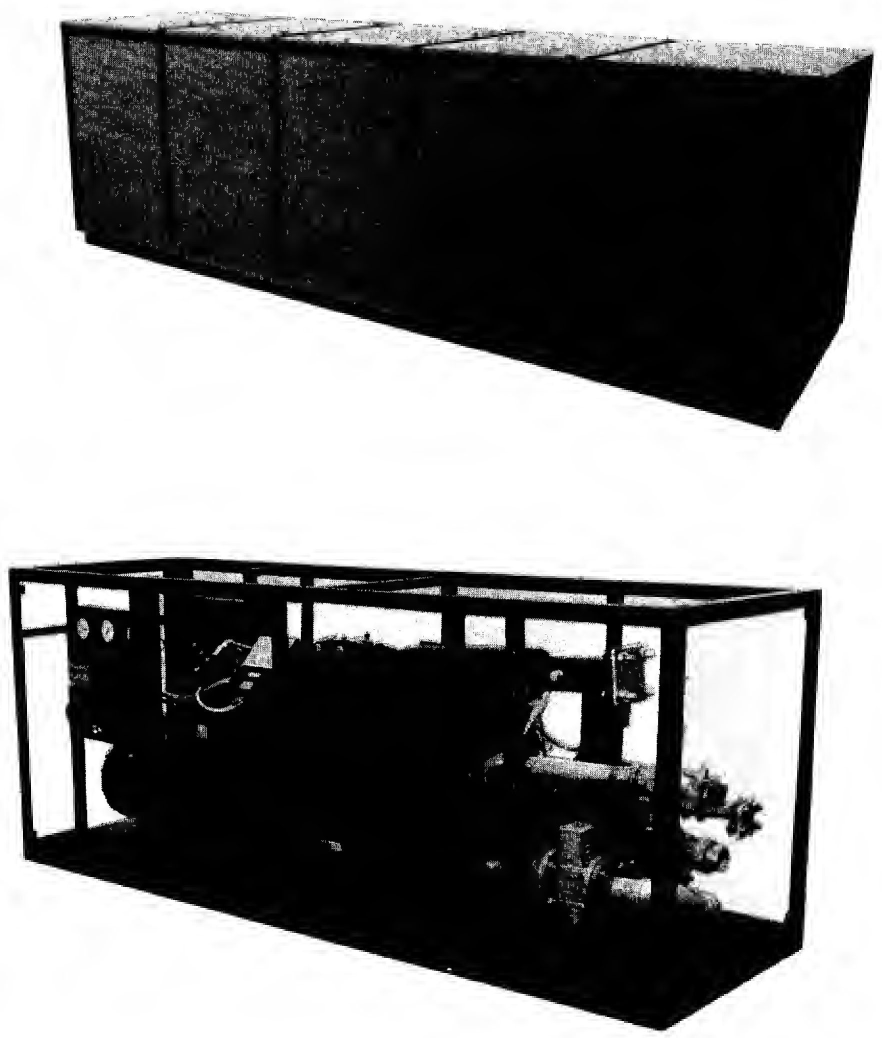


Figure 1-8. Condensing unit

POWER DISTRIBUTION UNITS

The mainframe, I/O Subsystem, and SSD all operate from 400 Hz 3-phase power. The mainframe, I/O Subsystem, and SSD have independent power distribution units.

The power distribution unit for the mainframe contains adjustable transformers for regulating the voltage to each column of the mainframe. The power distribution unit also contains temperature and voltage monitoring equipment that checks temperatures at strategic locations on the mainframe chassis. Automatic warning and shutdown circuitry protects the mainframe in case of overheating or excessive cooling. Control switches for the motor-generators and the condensing unit are mounted on the mainframe power distribution unit.

A smaller power distribution unit performs similar functions for the I/O Subsystem chassis or the SSD chassis.

Figure 1-9 shows the power distribution units for the mainframe (left) and for the I/O Subsystem or SSD (right).

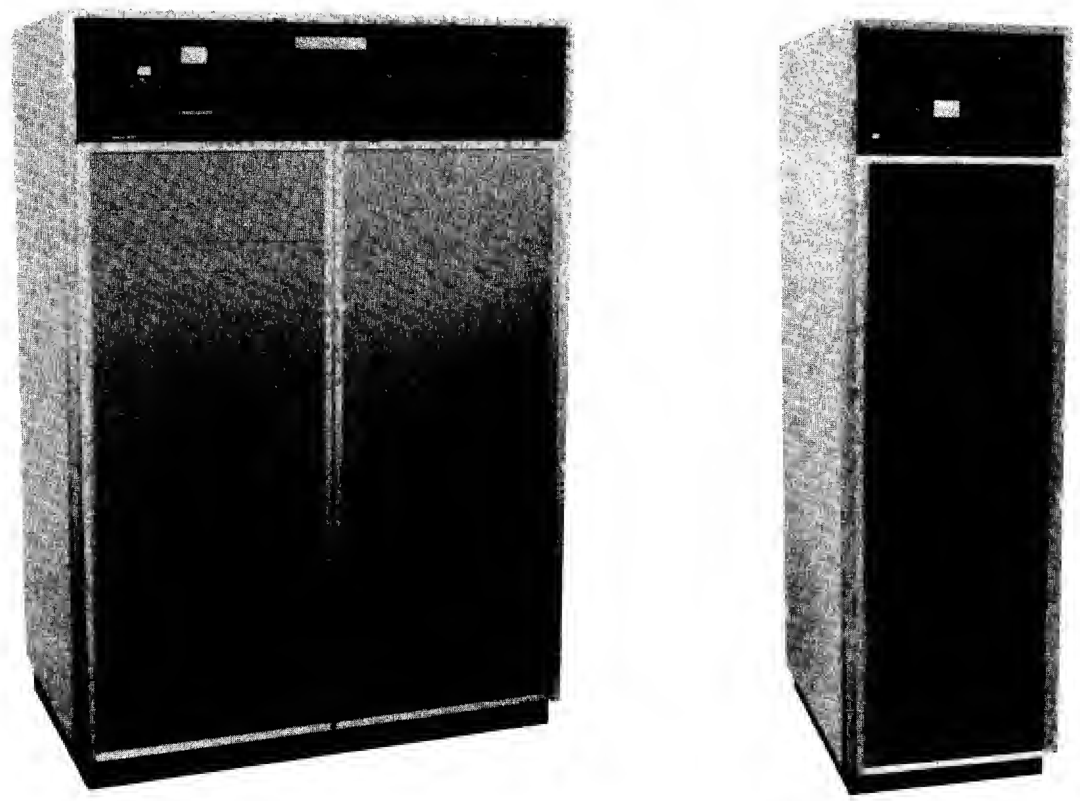


Figure 1-9. Power distribution units

MOTOR-GENERATOR UNITS

The motor-generator units convert primary power from the commercial power mains to the 400 Hz power used by the system. These units isolate the system from transients and fluctuations on the commercial power mains. The equipment consists of two or three motor-generator units and a control cabinet. Figure 1-10 shows a typical motor-generator and its control cabinet.

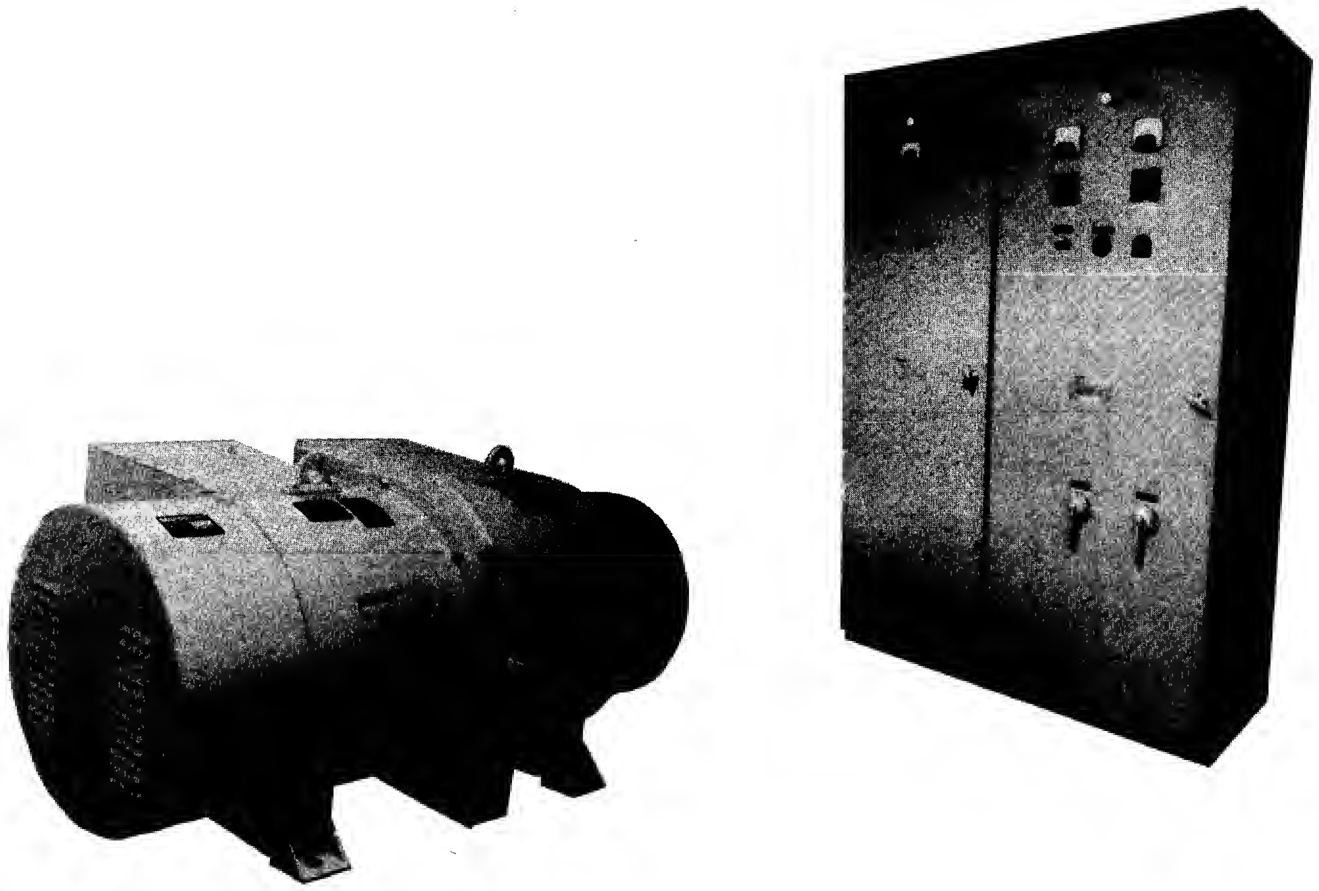


Figure 1-10. Motor-generator equipment

Figures 1-11 and 1-12 illustrate two configurations for the CRAY X-MP model 48.

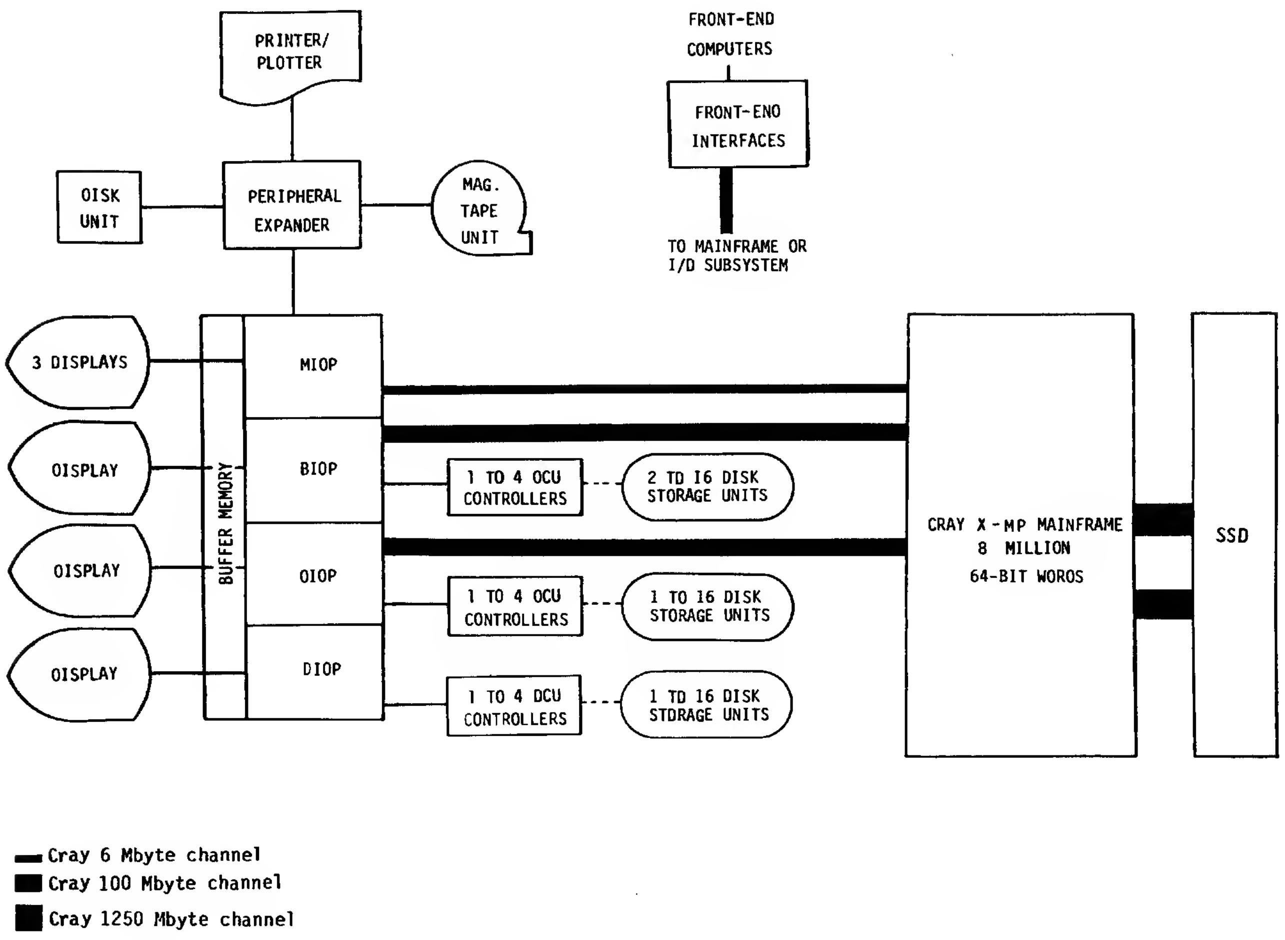


Figure 1-11. Block diagram of the 4-processor system with full disk capacity

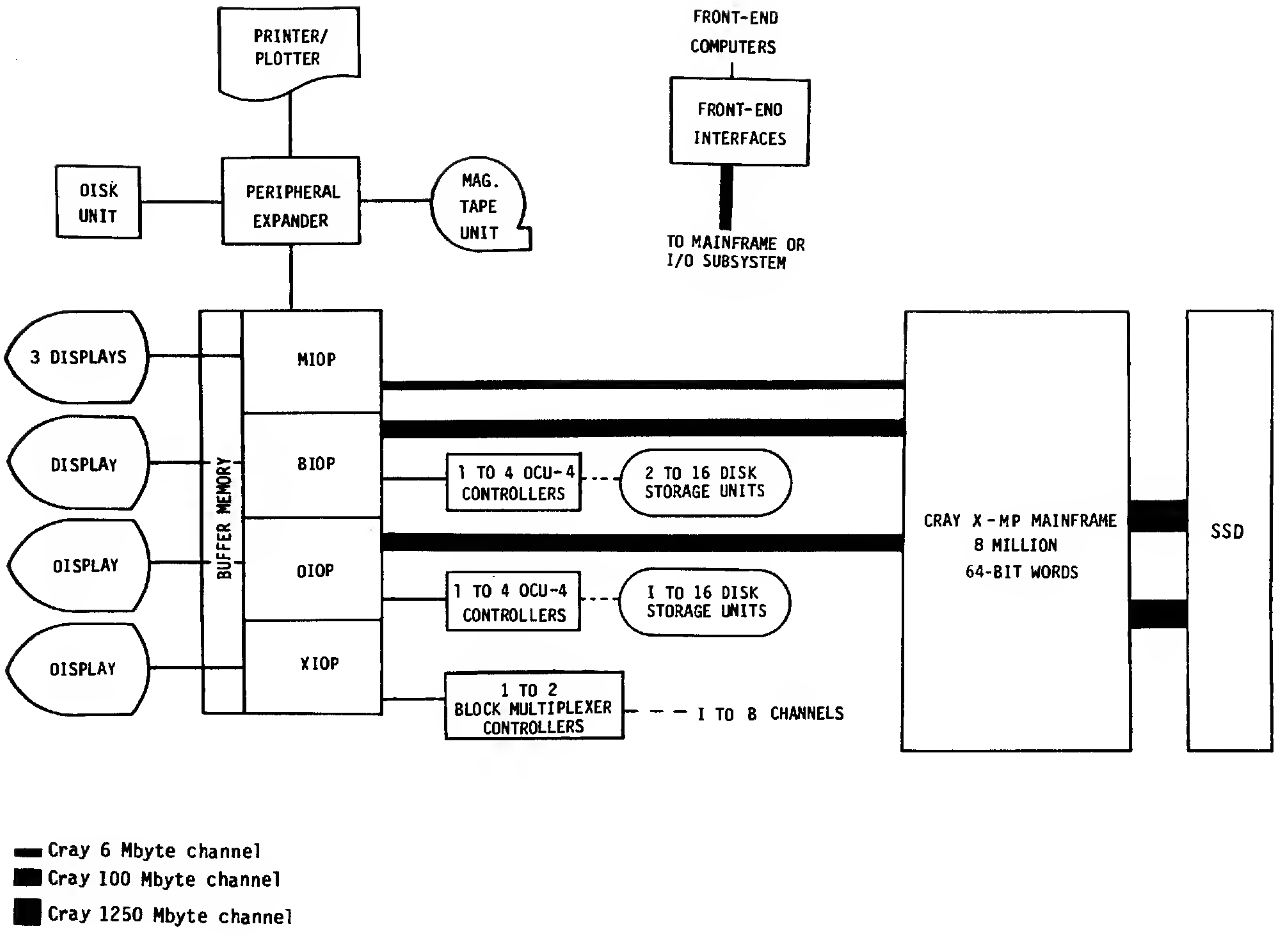


Figure 1-12. Block diagram of the 4-processor system with block multiplexer channels

CPU SHARED RESOURCES 2

INTRODUCTION

All four central processing units (CPUs) share the mainframe's Central Memory, the inter-CPU communication section, and the input/output section. These areas common to all CPUs are described in the following pages.

CENTRAL MEMORY

Central Memory consists of a number of banks of solid-state, random-access memory (RAM) and is shared by the CPUs and the I/O section. Standard Central Memory size for a 4-processor system is 8 million words with 64 banks. Banks are independent of each other. Each word is 72 bits with 64 data bits and 8 check bits. Sequentially addressed words reside in sequential banks.

Central Memory cycle time is 4 clock periods (CPs) or 38 nanoseconds (ns). Access time, the time required to fetch an operand from Central Memory to an operating register, is 14 CPs (152 ns) for A (address) and S (scalar) registers. Access time is 17 CPs + vector length for a V (vector) register and 16 CPs + block length for a block transfer to a B (intermediate address) or T (intermediate scalar) register.

The maximum transfer rate per CPU for B, T, and V registers is three words per CP; for A and S registers per CPU, it is one word every 2 CPs. Transfer of instructions to instruction buffers occurs at a rate of 32 parcels (8 words) per CP. For the I/O section, the transfer rate is 4 words per CP.

Central Memory features are summarized below and are described in detail in the following paragraphs.

- Shared access from all CPUs
- 8 million words of integrated circuit memory
- 64 data bits and 8 error correction bits per word
- 64 interleaved banks
- 4-CP bank cycle time

- Single error correction/double error detection (SECDED)
- 3 words per CP transfer rate to B, T, and V registers per CPU
- 1 word per 2 CP transfer rate to A and S registers per CPU
- 8 words per CP transfer rate to instruction buffers
- 4 words per CP transfer rate to I/O concurrent with all memory activity except instruction fetch and exchange

MEMORY ORGANIZATION

Memory is organized to provide fast, efficient access for all CPUs. Data transfers to and from memory are corrected with single error correction, double error detection. Central Memory is organized into four sections with 16 banks in each section.

Each CPU is connected to an independent access path into each of the four sections, as shown in figure 2-1. This configuration allows up to 16 memory references per clock period.

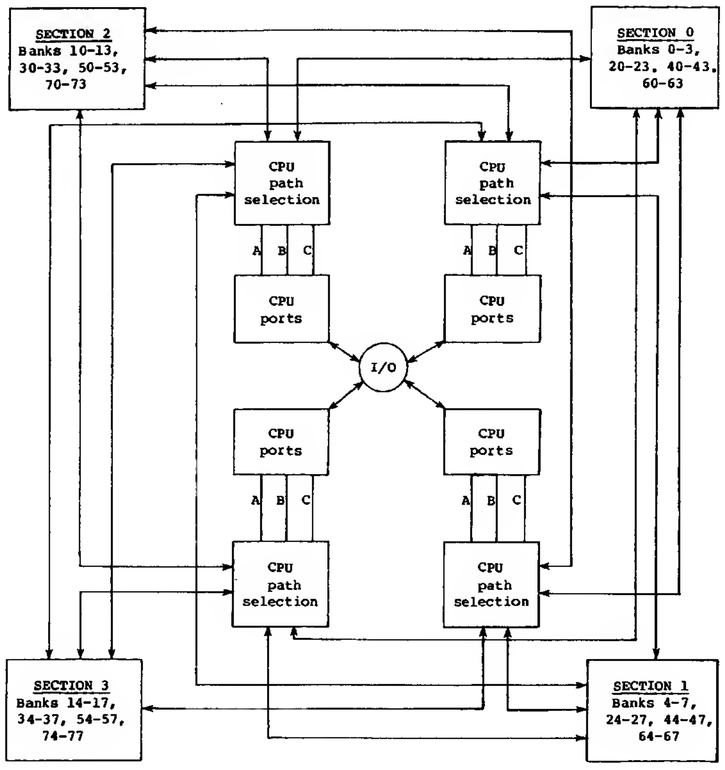

SECTION 2
Banks 10-13, 30-33, 50-53, 70-73
SECTION 0
Banks 0-3, 20-23, 40-43, 60-63
CPU path selection
A B C
CPU ports
I/O
SECTION 3
Banks 14-17, 34-37, 54-57, 74-77
SECTION 1
Banks 4-7, 24-27, 44-47, 64-67

Figure 2-1. Central Memory organization for a 4-processor system

MEMORY ADDRESSING

A word in a 64-bank memory is addressed in a maximum of 23 bits as shown in figure 2-2. The low-order 6 bits specify one of the 64 banks. The next 14-bit field specifies an address within the chip. The high-order 3 bits specify one chip on the module.

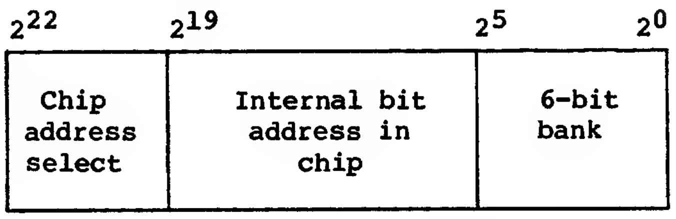


Figure 2-2. Memory address (64 banks)

MEMORY ACCESS

Each CPU in the system has four memory access ports, referred to as Port A, Port B, Port C, and I/O. Each port is capable of making one reference per CP. Ports A, B, and C are used for CPU register transfers.

B, T, and vector memory instructions issue to a particular memory port:

- Vector read (block reads only), B read instructions (176, 034) use Port A.
- Vector read (block reads only), T read instructions (176, 036) use Port B.
- Vector store, B, or T store instructions (177, 035, and 037) and scalar instructions (100-137) use Port C.

Once an instruction issues to a port, that port is reserved until all references are made for that instruction.

The references for each element of a block transfer (V,B,T) are made and completed in sequence through a port. However, since each reference is examined individually for possible conflicts, the data flow for a transfer may not be continuous. If an instruction requires a port that is busy, issue is blocked. Total execution time of the transfer depends on the number and type of conflicts encountered during the transfer.

****

CAUTION

Because concurrent block reads and writes are not examined for read before write or write before read (memory overlap hazard conditions), the software must detect where this condition occurs and ensure sequential operation.

****

The bidirectional memory mode enable (002500), bidirectional memory mode disable (002600), and the complete memory reference (002700) instructions are provided to resolve these cases and assure sequential operation. If the bidirectional memory mode is clear, block reads and writes are not allowed to operate concurrently within that CPU. Instruction 002700 allows the program to wait until the last references of all preceding block transfers are past the conflict resolution stage within the CPU issuing it and the transferred data is being transmitted to the designated memory or register locations. Instruction 002700 provides software a mechanism, wherever necessary in the program, to guarantee sequential memory operation within a CPU or between CPUs.

Issue of scalar memory references requires Ports A, B, and C to be available, ensuring sequential operation between block transfers and scalar references within a CPU.

A scalar reference conflict is detected in CP 4 of execution. If a conflict occurs, two more scalar references are allowed to issue. A fourth scalar reference holds issue if the conflict condition still exists for the first scalar reference.

Scalar references always execute in the order they are issued within a CPU. Instruction 002700 detects when all scalar references are past the conflict resolution stage within the CPU issuing it.

An I/O channel references memory through a specific CPU's I/O port (see subsection on CPU Input/Output Section). The I/O port can be active regardless of the activities on Ports A, B, or C.

For instruction fetches and exchange sequences, the CPUs are allowed access to memory in pairs; CPUs 0 and 1 comprise one pair, CPUs 2 and 3 another pair. Only one instruction fetch or exchange sequence can occur among the four CPUs at a time.

When a CPU requests an instruction fetch, referencing from all memory ports associated with that CPU pair is inhibited and the 32 banks being referenced are reserved (to prevent referencing from the other CPU pair). When memory is quiet (0 to 3 CPs), the fetch proceeds and references 32 banks in the next 4 CPs. Referencing of the eight ports is not enabled until 3 CPs later, to ensure all 32 banks are quiet.

NOTE

A fetch sequence that follows a scalar store can, under certain conditions, complete before the store. For this to happen, however, an out-of-buffer condition must arise before the scalar store is in CP 2 of execution. The out-of-buffer condition can occur before the scalar store is in CP 2 of execution if a buffer boundary is crossed without doing a branch. This presents a problem only if the fetch and store are to the same area in memory. Therefore, software that utilizes dynamic coding should ensure that the code generated is actually in memory before that area of memory is fetched into the instruction buffers.

During this time, the other CPU pair has access to the remaining banks of memory.

When a CPU requests an exchange, all referencing from the four memory ports of the other CPU in the CPU pair is inhibited and 32 banks are reserved (to prevent referencing from the other CPU pair). When memory is quiet (0 to 3 CPs), the exchange proceeds and references 16 banks in the next 20 CPs. Each bank is referenced twice during this time, once for a read and once for a write. An exchange sequence requires all activities within a CPU to complete before the exchange request is made. As with the instruction fetch, the other CPU pair has access to the remaining banks of memory.

A fetch request follows immediately after the exchange is complete and then referencing from the memory ports of the other CPU in the pair is enabled.

Conflict resolution

During each clock period, references to the memory ports in the system are examined for memory access conflicts. If a conflict occurs for a reference, the reference is held and no further referencing from that port is allowed until the conflict is resolved.

Three types of memory access conflicts can occur: Bank Busy, Simultaneous Bank, and Section Access.

Bank Busy conflict - The Bank Busy conflict is caused by any port within or between CPUs requesting a bank currently in a reference cycle. Resolution of this conflict occurs when the bank cycle is complete. All ports in the CPU are held 1, 2, or 3 CPs because of a Bank Busy conflict.

Simultaneous Bank conflict - The Simultaneous Bank conflict is caused by two or more ports in different CPUs requesting the same bank. Resolution of this conflict is based on a priority (see subsection below on Memory access priorities). All ports in a CPU are held 1 CP because of a Simultaneous Bank conflict. A Bank Busy conflict always follows a Simultaneous Bank conflict.

Section Access conflict - The Section Access conflict is caused by two or more ports in the same CPU requesting any bank in the same section. Resolution of this conflict is based on priority. The highest priority port is allowed to proceed, all other ports involved in this conflict hold (see subsection below on Memory access priorities). The port is held 1 CP because of a section access conflict.

Memory access priorities

The following priorities are used to resolve memory access conflicts.

- Intra-CPU priority: the priority between Ports A, B, and C is determined by the following conditions:
  - Any port with an odd increment always has a higher priority than a port with an even increment, regardless of their issued sequence.
  - Among all ports with the same type of increment (odd or even), the relative time of issue determines the priority, with the first issued having the highest priority.
- Inter-CPU priority: every 4 CPs the priority between CPUs changes.
- I/O priority: the I/O ports are always lowest priority, within CPUs.

MEMORY ERROR CORRECTION

A single error correction/double error detection (SECDED) network is used between a CPU and memory. SECDED assures that data written into memory can be returned to the CPU with consistent precision (figure 2-3).

If a single bit of a data word is altered, the single error alteration is automatically corrected before passing the data word to the computer. If 2 bits of the same data word are altered, the error is detected but not corrected. In either case, the CPU can be interrupted, depending on interrupt options selected to allow processing of the error. For 3 or more bits in error, results are ambiguous.

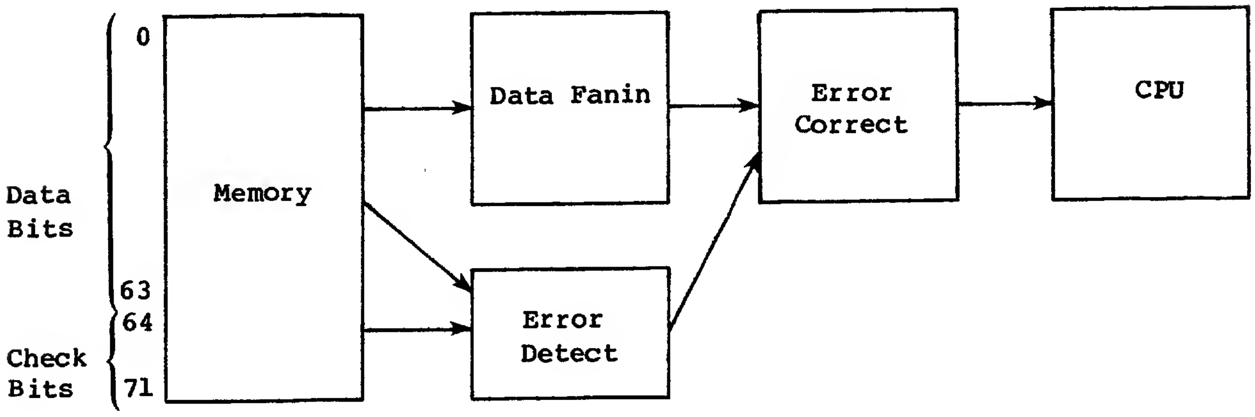


Figure 2-3. Memory data path with SECDED

The SECDED error processing scheme is based on error detection and correction codes devised by R. W. Hamming.† An 8-bit check byte is appended to the 64-bit data word before the data is written in memory. The 8 check bits are generated as even parity bits for a specific group of data bits. Figure 2-4 shows the bits of the data word used to determine the state of each check bit. An X in the horizontal row indicates that data bit contributes to the generation of that check bit. Thus, check bit 0 is the bit that makes group parity even for the group of bits 2^1, 2^3, 2^5, 2^7, 2^9, 2^{11}, 2^{13}, 2^{15}, 2^{17}, 2^{19}, 2^{21}, 2^{23}, 2^{25}, 2^{27}, 2^{29}, and 2^{31} through 2^{55}.

The 8 check bits and the data word are stored in memory at the same location. When read from memory, the same 64-bit matrix of figure 2-4 is used to generate a new set of check bits, which are compared with the old check bits. The resulting 8 comparison bits are called syndrome†† bits (S bits). The states of these S bits are all symptoms of any error that occurred (1=No compare). If all syndrome bits are 0, no memory error is assumed.

† Hamming, R.W., "Error Detection and Correcting Codes," Bell System Technical Journal, 29, No. 2, pp. 147-160 (April, 1950).

†† Syndrome: Any set of characteristics regarded as identifying a certain type, condition, etc. Webster's New World Dictionary.

CHECK BYTE

	2^{71}	2^{70}	2^{69}	2^{68}	2^{67}	2^{66}	2^{65}	2^{64}
check bit 0								x
check bit 1							x	
check bit 2						x		
check bit 3					x			
check bit 4				x				
check bit 5			x					
check bit 6		x						
check bit 7	x							

	2^{63}	2^{62}	2^{61}	2^{60}	2^{59}	2^{58}	2^{57}	2^{56}
check bit 0								
check bit 1	x	x	x	x	x	x	x	x
check bit 2	x	x	x	x	x	x	x	x
check bit 3	x	x	x	x	x	x	x	x
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{55}	2^{54}	2^{53}	2^{52}	2^{51}	2^{50}	2^{49}	2^{48}
check bit 0	x	x	x	x	x	x	x	x
check bit 1								
check bit 2	x	x	x	x	x	x	x	x
check bit 3	x	x	x	x	x	x	x	x
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{47}	2^{46}	2^{45}	2^{44}	2^{43}	2^{42}	2^{41}	2^{40}
check bit 0	x	x	x	x	x	x	x	x
check bit 1	x	x	x	x	x	x	x	x
check bit 2								
check bit 3	x	x	x	x	x	x	x	x
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{39}	2^{38}	2^{37}	2^{36}	2^{35}	2^{34}	2^{33}	2^{32}
check bit 0	x	x	x	x	x	x	x	x
check bit 1	x	x	x	x	x	x	x	x
check bit 2	x	x	x	x	x	x	x	x
check bit 3								
check bit 4	x		x		x		x	
check bit 5	x	x			x	x		
check bit 6	x	x	x	x				
check bit 7	x			x		x	x	

	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	2^{24}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4								
check bit 5	x	x	x	x	x	x	x	x
check bit 6	x	x	x	x	x	x	x	x
check bit 7	x	x	x	x	x	x	x	x

	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4	x	x	x	x	x	x	x	x
check bit 5								
check bit 6	x	x	x	x	x	x	x	x
check bit 7	x	x	x	x	x	x	x	x

	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4	x	x	x	x	x	x	x	x
check bit 5	x	x	x	x	x	x	x	x
check bit 6								
check bit 7	x	x	x	x	x	x	x	x

	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
check bit 0	x		x		x		x	
check bit 1	x	x			x	x		
check bit 2	x	x	x	x				
check bit 3	x			x		x	x	
check bit 4	x	x	x	x	x	x	x	x
check bit 5	x	x	x	x	x	x	x	x
check bit 6	x	x	x	x	x	x	x	x
check bit 7								

Figure 2-4. Error correction matrix

Any change of state of a single bit in memory causes an odd number of syndrome bits to be set to 1. A double error (an error in 2 bits) appears as an even number of syndrome bits set to 1.

The matrix is designed so that:

- If all syndrome bits are 0, no error is assumed.
- If only 1 syndrome bit is 1, the associated check bit is in error.
- If more than 1 syndrome bit is 1 and the parity of syndrome bits S0 through S7 is even, then a double error (or an even number of bit errors) occurred within the data bits or check bits.

- If more than 1 syndrome bit is 1 and the parity of all syndrome bits is odd, then a single and correctable error is assumed to have occurred. The syndrome bits can be decoded to identify the bit in error.

- If 3 or more memory bits are in error, the parity of all syndrome bits is odd and results are ambiguous.

Modules involved with generating and interpreting the 8-bit check byte used for SECDED include logic that can be used for verifying check bit storage, check bit generation, and error detection and correction. Refer to Appendix D for information on SECDED maintenance functions.

INTER-CPU COMMUNICATION SECTION

The inter-CPU communication section of the system contains special hardware for communication among the CPUs, for control, and for a real-time clock. The Real-time Clock (RTC), Shared Address (SB), Shared Scalar (ST), and Semaphore (SM) registers are shared by the CPUs. These registers with their sources and destinations are shown in figure 2-5 and described in the following paragraphs.

REAL-TIME CLOCK

The mainframe contains one Real-time Clock (RTC) register shared by the CPUs. Programs can be timed precisely by using the clock period (CP) counter. This counter is 64 bits wide and advances one count each CP of 9.5 nanoseconds. Since the clock advances synchronously with program execution, it can be used to time the program to an exact number of CPs. However, in such an application, the counting can contain counts from other tasks if an interrupt occurs before the end time is read.

Instructions used with the RTC register are:

0014*j*0	RT S*j*	Enter the RTC register with (S*j*)
072*i*00	S*i* RT	Transmit (RTC) to S*i*

A program reads the CP counter using instruction 072 and resets it with instruction 0014*j*0. Loading or reading the CP counter can occur from all CPUs at the same time. If more than one CPU is in monitor mode, the software should ensure that only one CPU enters a value into this register.

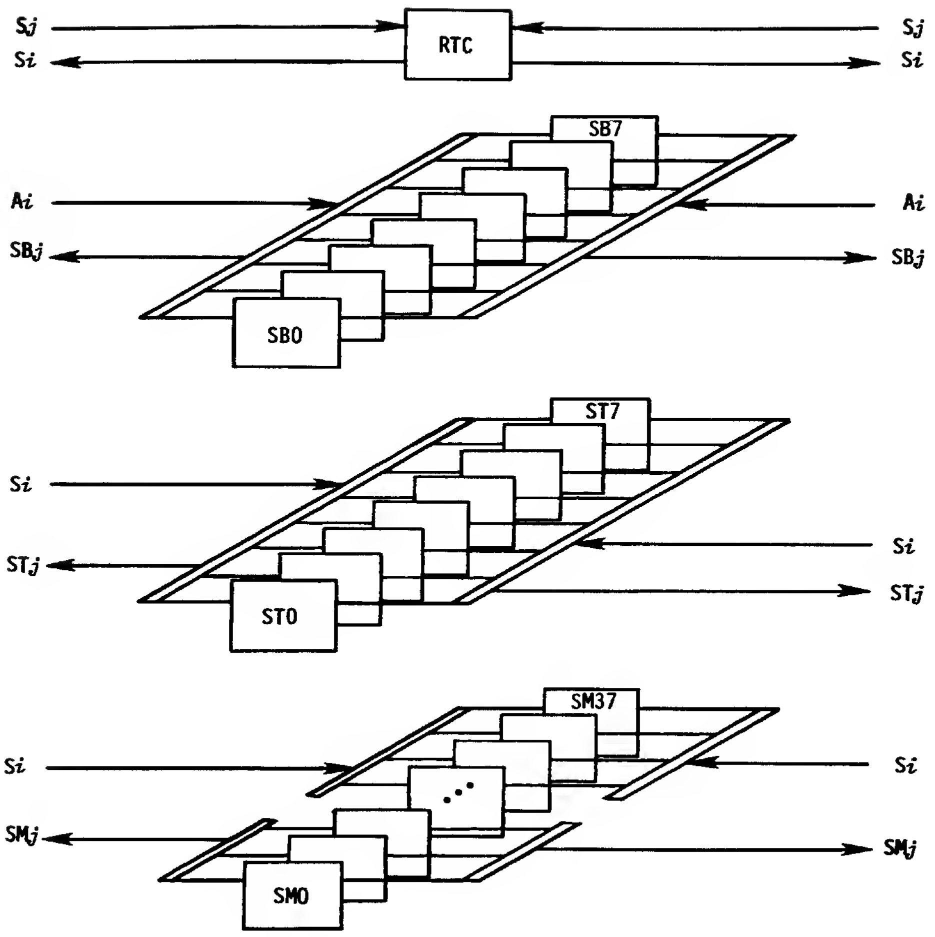


Figure 2-5. Shared registers and real-time clock

INTER-CPU COMMUNICATION AND CONTROL

Five identical sets of shared registers are used for communication and control among CPUs. Each set contains eight 24-bit Shared Address (SB) registers, eight 64-bit Shared Scalar (ST) registers and 32 1-bit Semaphore (SM) registers.

Each CPU's Cluster Number (CLN) register determines which set of shared registers is accessed by a CPU (clustering). The CLN register is loaded from the Exchange Package or, if the CPU is in monitor mode, through instruction 0014*j*3.

The CLN register can contain one of six different values. Values 1, 2, 3, 4, or 5 allow the CPU to access one of the five sets of shared registers. Value 0 prevents any access to shared registers by the CPU. If the value is 0, instructions regarding the shared registers become no-ops, except for the instructions returning values to A*i* or S*i*, which return a zero value. If the CLN registers in more than one CPU are set to the same value (1, 2, 3, 4, or 5), then those CPUs share a common set of SB, ST, and SM registers.

Shared Address and Shared Scalar registers

The Shared Address (SB) and Shared Scalar (ST) registers are used for passing address and scalar information from one CPU to another. No hardware reservations are made on these registers. Any necessary reservations to restrict access to these registers must be handled in the software through use of the Semaphore (SM) registers or by shared memory design. The single hardware restriction on access to the SB and ST registers is that only one read or one write operation can occur in a CP.

The instructions used with the SB and ST registers are:

026*ij*7	A*i* SB*j*	Transmit (SB*j*) to A*i*
027*ij*7	SB*j* A*i*	Transmit (A*i*) to SB*j*
072*ij*3	S*i* ST*j*	Transmit (ST*j*) to S*i*
073*ij*3	ST*j* S*i*	Transmit (S*i*) to ST*j*

Semaphore registers

The Semaphore (SM) registers are used for control among the CPUs. No hardware reservations are made on these registers. Loading or reading the SM registers or setting or clearing a particular SM register can occur at any time from any or all CPUs.

The test and set instruction (0034*jk*) is the only operation on the SM registers including a hardware interlock. This interlock prevents a simultaneous test and set operation on the same SM register from more than one CPU. The test and set instruction first tests the value of the selected SM register. If the value is 0, the instruction issues and sets that SM register to a 1. If the value is 1, the instruction holds issue until the value is 0.

When all CPUs in a cluster are holding issue on a test and set instruction, a deadlock interrupt can occur. All CPUs with equal cluster numbers above 0 belong to the same cluster and must be holding issue on a test and set instruction to cause a deadlock interrupt. When that happens, all CPUs in the cluster receive deadlock interrupts. If only one CPU belongs to a cluster and holds issue on a test and set instruction, that CPU receives a deadlock interrupt. No deadlock interrupt can occur in cluster 0 (CLN=0).

When an interrupt occurs, normally the instructions already in the NIP and CIP registers are allowed to issue before the exchange sequence starts. If a test and set instruction is holding in the CIP register and an interrupt occurs, a special exchange start-up sequence is initiated. In this case, the instruction in the NIP register and the test and set instruction in the CIP register are discarded and the Program Counter (P) register is adjusted to point to the discarded test and set instruction. The Waiting on Semaphore (WS) flag in the Exchange Package sets, indicating a test and set instruction was holding in the CIP register when the interrupt occurred. The exchange sequence is then started.

Instructions used with the SM registers are:

0034*jk*	SM*jk*	1,TS	Test and set, SM*jk*
0036*jk*	SM*jk*	0	Clear SM*jk*
0037*jk*	SM*jk*	1	Set SM*jk*
072*i*02	S*i*	SM	Transmit (SM) to S*i*
073*i*02	SM	S*i*	Transmit (S*i*) to SM

Shared register and semaphore conflicts

? fixed priority or round robin

A scanner is used to break a tie caused by simultaneous requests for access to the Semaphores or Shared registers of any cluster. If there is no competition for access, no extra hold issues are generated. For example, an 027*ij*7 holds issue 3 CP, but if there is an access conflict, issue holds until a scanner with four slots breaks the tie. A request takes 2 CPs to complete; therefore, subsequent requests can be accepted every other CP until all requests are resolved.

CPU INPUT/OUTPUT SECTION

The input/output section of the mainframe is shared by all Central Processing Units (CPUs). The mainframe supports three channel types identified by their maximum transfer rates of 1250 Mbytes per second, 100 Mbytes per second, and 6 Mbytes per second.

Two 1250 Mbyte per second channel pairs transfer data between Central Memory and a Solid-state Storage Device (SSD). These channels are 128 bits wide and use 16 check bits in each direction. A maximum transfer rate of over 10 gigabits per second is possible on a 1250 Mbyte per second channel. The channel is two parallel 64-bit channels each with SECDED; therefore, under certain circumstances the full-width channel can correct double errors.

Four 100 Mbyte per second channel pairs transfer data between Central Memory and an I/O Subsystem. A 100 Mbyte per second channel is 64 bits wide and uses 8 check bits in each direction. Data words are transferred in blocks of 16 under control of Data Ready and Data Transmit control signals. Each 100 Mbyte per second channel has a maximum transfer rate of approximately 850 Mbits per second.

I/O Subsystem communication with the CPUs is over four pairs of control channels, each with a maximum transfer rate of 6 Mbytes per second. Each 6 Mbyte per second channel is 16 bits wide.

All I/O (including 100 Mbyte and 1250 Mbyte per second channels) uses the I/O ports to memory. Access to these ports is controlled by a scanner. All CPU memory ports (Ports A, B, and C) have higher priority than the I/O ports.

Channel features of the input/output section are summarized below and described in the remainder of this section.

- Two channel pairs with 1250 Mbytes per second maximum transfer rate per channel
  - 128 data bits and 16 check bits in each direction
- Four channel pairs with 100 Mbytes per second maximum transfer rate per channel
  - 64 data bits, 3 control bits, and 8 check bits in each direction
- Four I/O channel pairs, 6 Mbytes per second maximum transfer rate per channel
  - Shared control from the CPUs
  - 16 data bits, 3 control bits, and 4 parity bits in each direction
  - Lost data detection
- Channels are divided into four groups; each group contains either input or output channels.
- Channel groups are served equally by memory (each group is scanned every 4 CPs).
- Channel priority is resolved within channel groups.

DATA TRANSFER FOR SOLID-STATE STORAGE DEVICE

Data is transferred directly between the Solid-state Storage Device (SSD) and the mainframe using the 1250 Mbyte per second channels. A 1250 Mbyte per second channel is 128 bits wide and is programmed through software. Programming details for the SSD are described in the Solid-state Storage Device (SSD) Reference Manual, CRI publication HR-0031.

DATA TRANSFER FOR I/O SUBSYSTEM

A 100 Mbyte per second channel pair transfers data between Central Memory and the Buffer I/O Processor (BIOP) of the I/O Subsystem. A second 100 Mbyte per second channel pair transfers data between Central Memory and a Disk I/O Processor (DIOP) or Auxiliary I/O Processor (XIOP).† Each channel is 64 bits wide and handles data at approximately 100 Mbytes per second. Each channel uses an additional 8 check bits for single error correction/double error detection (SECDED), as is used in Central Memory.

The CPU side of a 100 Mbyte per second channel pair uses a pair of 16-word buffers to stream the data out of Central Memory and another pair to stream data into Central Memory. On output, as one buffer block is being sent to the I/O Processor (IOP), the other buffer is filling from Central Memory. Similarly, on input, one buffer block is filling from an IOP while the other is transmitting to Central Memory.

At the IOP side of a 100 Mbyte per second channel pair, data passing into Local Memory (an I/O Processor's memory) is double buffered and disassembled into 16-bit parcels. The channel side passing data from Local Memory simply assembles 16-bit parcels into 64-bit words for transmission to a CPU.

An I/O Processor controls a 100 Mbyte per second channel pair linking it with Central Memory. The IOP initiates all data transfers on the channel and performs all error processing required for the channel. There are no CPU instructions for the 100 Mbyte per second channel pair. Programming details for the 100 Mbyte per second channels are contained in the I/O Subsystem Reference Manual, CRI publication HR-0030.

6 MBYTE PER SECOND CHANNELS

Standard control channels for the system are 6 Mbyte per second channels. Each 6 Mbyte per second channel has 16-bit asynchronous control logic used for front-end interfaces. The instructions used with 6 Mbyte per second channels follow.

† Software does not currently support data transfer using the 100 Mbyte per second channel pair to an XIOP.

0010*jk*	CA,A*j* A*k*	Set the Current Address (CA) register for the channel indicated by (A*j*) to (A*k*) and activate the channel.
0011*jk*	CL,A*j* A*k*	Set the Limit Address (CL) register for the channel indicated by (A*j*) to (A*k*).
0012*jk*	CI,A*j*	Clear the Interrupt flag and Error flag for the channel indicated by (A*j*): Output channel *k*=0; clear MC, *k*=1; set MC. Input channel *k*=0; no operation, *k*=1; clear held ready.
033*i*00	A*i* CI	Transmit channel number to A*i*.
033*ij*0	A*i* CA,A*j*	Transmit address of channel (A*j*) to A*i*.
033*ij*1	A*i* CE,A*j*	Transmit Error flag of channel (A*j*) to A*i*.

MULTI-CPU PROGRAMMING

The 6 Mbyte per second I/O channels can operate from any CPU, and any CPU can issue instructions to any of the channels. There is no hardware interlock among the CPUs; therefore, software must ensure that only one CPU is servicing I/O at a time, while in monitor mode. Instruction 033 is independent in nature and can be issued without an interlock.

The following conditions must be met for an I/O interrupt to occur.

- No CPU waiting for an exchange
- No CPU in monitor mode
- An interrupt is present

Normally, the interrupt from a 6 Mbyte per second channel is directed toward the CPU that last issued a clear interrupt instruction (0012) to that channel. However, because an I/O interrupt occurs in only one CPU at a time, the following conditions (in priority order) determine the CPU toward which the interrupt is directed. Once in monitor mode, a CPU should service all I/O interrupts.

1. All I/O interrupts are directed toward a CPU that has the select external interrupt mode set.

2. If no CPU has selected external interrupts, then interrupts are directed toward a CPU holding issue on a test and set instruction.

3. If neither conditions 1 nor 2 exist or if they exist in all CPUs, the interrupt is directed to the CPU that last issued a clear interrupt instruction to that channel.

6 MBYTE PER SECOND CHANNEL OPERATION

Input and output channels access Central Memory directly. Input channels store external data in memory and output channels read data from memory. A primary task of a channel is to convert 64-bit Central Memory words into 16-bit parcels or 16-bit parcels into 64-bit Central Memory words. Four parcels make up one Central Memory word with bits of the parcels assigned to memory bit positions as shown in table 2-1. In both input and output operations, parcel 0 is always transferred first.

Each input or output channel has a data channel (4 parity bits, 16 data bits, and 3 control lines), a 64-bit assembly or disassembly register, a channel Current Address (CA) register, and a channel Limit Address (CL) register.

Three control signals (Ready, Resume, and Disconnect) coordinate the transfer of parcels over the channels. In addition to the three control signals, the output channel of a pair has a Master Clear line. Appendix B describes the signal sequence of a 6 Mbyte per second channel.

I/O interrupts can be caused by the following:

- On all output channels, if (CA) becomes equal to (CL), then the resume for the last parcel transmitted sets interrupt.
- External device disconnect is received on any input channel and channel is active.
- Channel error condition occurs (described later in this section).

The number of the channel causing an interrupt can be determined by using instruction 033, which reads into A*i* the highest priority channel number requesting an interrupt. The lowest numbered channel has the highest priority. The interrupt request continues until cleared by the monitor program when an interrupt from the next highest priority channel, if present, is sensed. All interrupts are available through instruction 033 to all CPUs. Channel numbers for 6 Mbyte per second channels are 10_8 through 17_8 (10/11, 12/13, 14/15, and 16/17 - even for input, odd for output).

INPUT CHANNEL PROGRAMMING

To start an input operation, the CPU program (see figure 2-6):

1. Sets the channel limit address to the last word address + 1 (LWA+1), and
2. Sets the channel current address to the first word address (FWA).

Table 2-1. Channel word assembly/disassembly

Characteristic	Bit position	Number of bits	Comment
Channel data bits	$2^{15}-2^{0}$	16	Four 4-bit groups
Channel parity bits		4	One per 4-bit group
CRAY X-MP word	$2^{63}-2^{0}$	64	
Parcel 0	$2^{63}-2^{48}$	16	First in or out
Parcel 1	$2^{47}-2^{32}$	16	Second in or out
Parcel 2	$2^{31}-2^{16}$	16	Third in or out
Parcel 3	$2^{15}-2^{0}$	16	Fourth in or out

Setting the current address causes the Channel Active flag to set. The channel is then ready to receive data. When a 4-parcel word is assembled, the word is stored in memory at the address contained in the CA register. When the word is accepted by memory, the current address is advanced by 1.

An external transmitting device sends a Disconnect signal to indicate end of a transfer. When the Disconnect signal is received, the Channel Interrupt flag sets and a test is performed to check for a partially assembled word. If the partial word is found, the valid portion of the word is stored in memory and the unreceived, low-order parcels are stored as zeros.

The Interrupt flag sets when a Disconnect signal is received or when the Channel Error flag is set.

INPUT CHANNEL ERROR CONDITIONS

Input channel error conditions can occur at a parcel level (parity error). When a parcel in error occurs, the Parity Fault flag sets immediately. The Parity Fault flag does not generate an interrupt, it is saved and sets the Error flag when a disconnect occurs. Therefore, the program should check the state of the Error flag when an interrupt is honored. All parcels stored after the error are zeroed.

If a Ready signal is received when the channel is not active, the Ready condition is held until the channel is activated. At this time, a Resume signal is sent. No Error flag is set and no interrupt request is generated. Since the Ready condition is held when the channel is inactive, it is sometimes advantageous to be able to clear this Ready signal before setting up the channel, especially on a deadstart or a

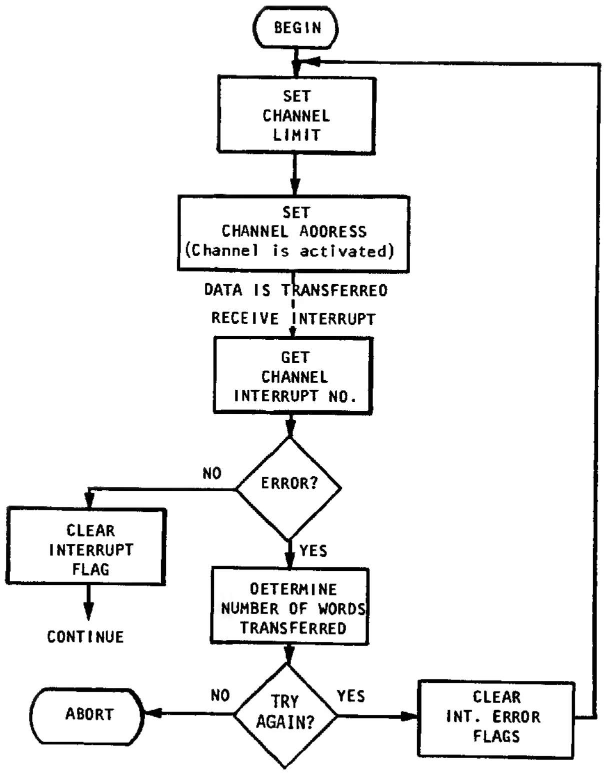


Figure 2-6. Basic I/O program flowchart

resynchronization of the channel after an error. The Ready signal can be cleared by using instruction 0012*j*1 to input channel (A*i*), clearing any Ready signal being held before issue of instruction 0012*j*1.

OUTPUT CHANNEL PROGRAMMING

To start an output operation, the CPU program:

1. Sets the channel limit address to the last word address + 1 (LWA+1), and

2. Sets the channel current address to the first word address (FWA).

Setting the current address causes the Channel Active flag to be set. The channel reads the first word from memory addressed by the contents of the CA register. When the word is received from memory, the channel advances the current address by 1 and starts the data transfer.

After each word is read from memory and the current address is advanced, the limit test is made, comparing the contents of the CA register and the CL register. If they are equal, the operation is complete as soon as the last parcel transfer is finished.

The Interrupt flag also sets if an error is detected. The only error that an output channel detects is a Resume signal received when the channel is inactive. No external response is generated.

PROGRAMMED MASTER CLEAR TO EXTERNAL DEVICE

The system can send a Master Clear signal to an external device through the output channel. The external Master Clear sequence is as follows.

1.	0012*jk*	Clears input channel to ensure external activity on the channel pair has stopped
2.	0012*j*1	Clears output channel to ensure CPU activity on the channel pair has stopped. Set Master Clear.
3.	Delay 1	Device dependent; determines the duration of the Master Clear signal.
4.	0012*j*0	Clears the output channel. This turns off the Master Clear signal.
5.	Delay 2	Device dependent; allows time for initialization activities in the attached device to complete.

For Cray Research, Inc., front-end interfaces, delays 1 and 2 should each be a minimum of 80 CPs.

ACCESS TO CENTRAL MEMORY

Each CPU has one I/O port to memory. Channels are divided into four groups and scanned to allow access to memory. Each of the four channel groups shown below is assigned a time slot (figure 2-7) that is scanned for a memory request once every 4 CPs. The channel listed first in each group has the highest priority. During the next 3 CPs, the scanner allows requests from the other three channel groups. Therefore, an I/O memory request can occur every CP. The scanner stops for all memory conflicts caused by an I/O reference and also stops for a block (100 Mbyte per second channel) reference while a buffer is referencing, maximum 16 words (figure 2-8).

Channels A, B, C, and D are 100 Mbyte per second channels. Channels 6 and 7 are 1250 Mbyte per second channels. Channels 10 through 17 are 6 Mbyte per second channels.

		CPU			
		0	1	2	3
Group	0 (input)	A, 10	B, 14	C	D
	1 (output)	A, 11	B, 15	C	D
	2 (input)	7, 12	7, 16	6	6
	3 (output)	7, 13	7, 17	6	6

I/O LOCKOUT

An I/O memory request can be locked out by an exchange sequence or instruction fetch sequence.

MEMORY BANK CONFLICTS

Memory bank conflicts are tested for CPU scalar, vector, and I/O memory references. When an exchange sequence or instruction fetch sequence is in progress, all other memory references for the CPU pair are locked out.

Each memory bank can accept a new request every 4 CPs. To test for a memory bank conflict, the 6 low-order bits of the memory address are checked against Bank Busy conflicts and other memory references. The bank is busy for 4 CPs on a reference.

I/O MEMORY CONFLICTS

Before testing for a memory bank conflict, a check is made to ensure no exchange sequence or instruction fetch sequence is in progress. If either of these conditions exists, the I/O request is held. The 6 low-order address bits are tested against Bank Busy conflicts and other memory references. If a bank being referenced is busy, the reference is held and the scanner is stopped.

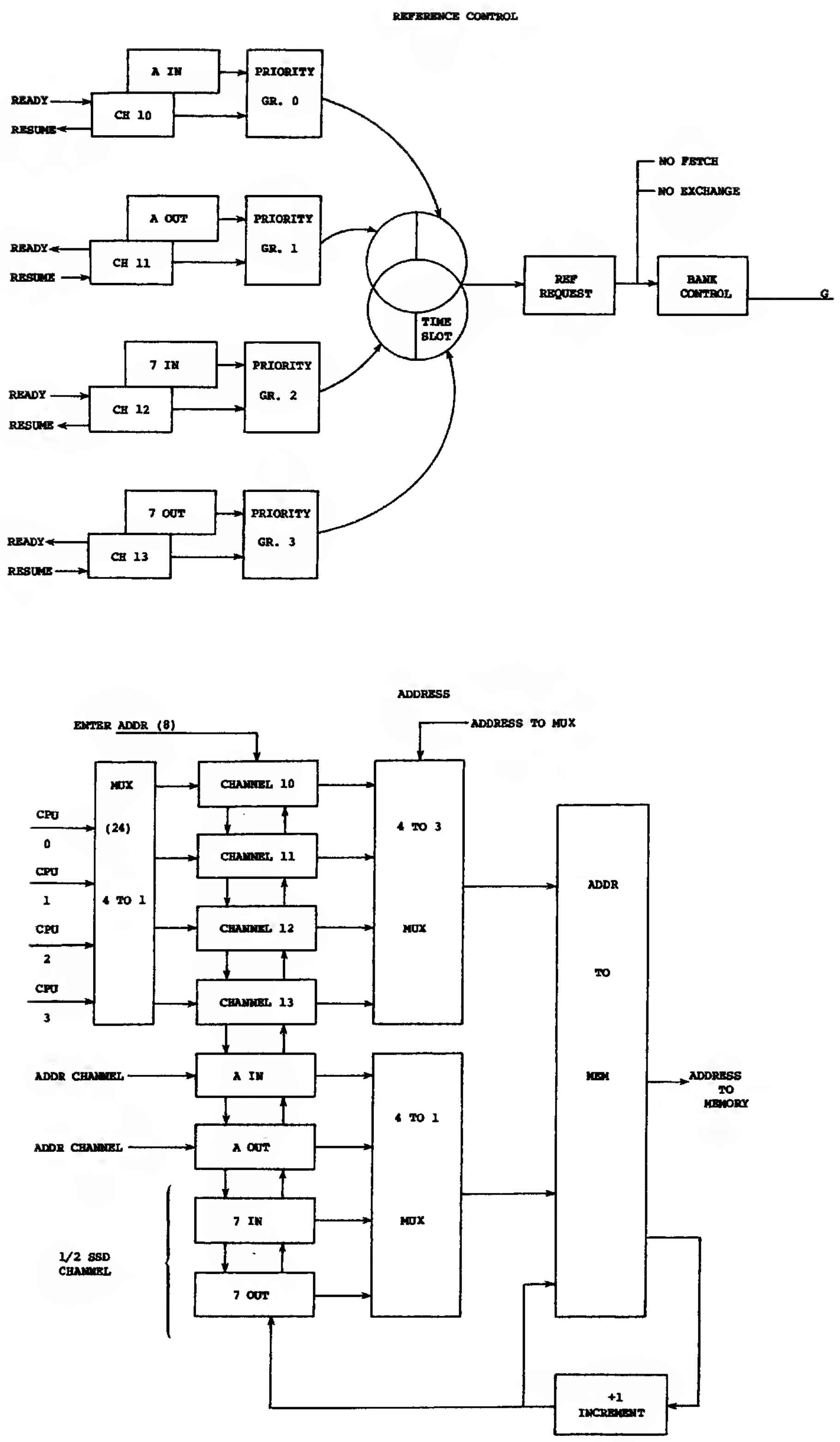
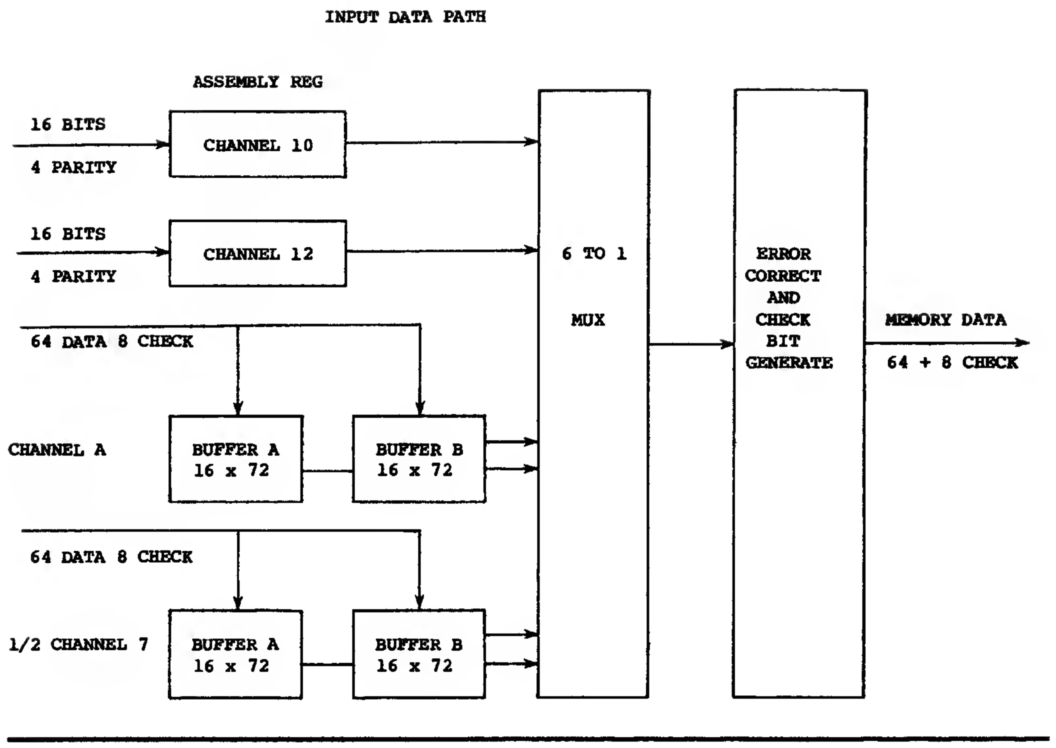


Figure 2-7. Channel I/O control (shown for CPU 0)



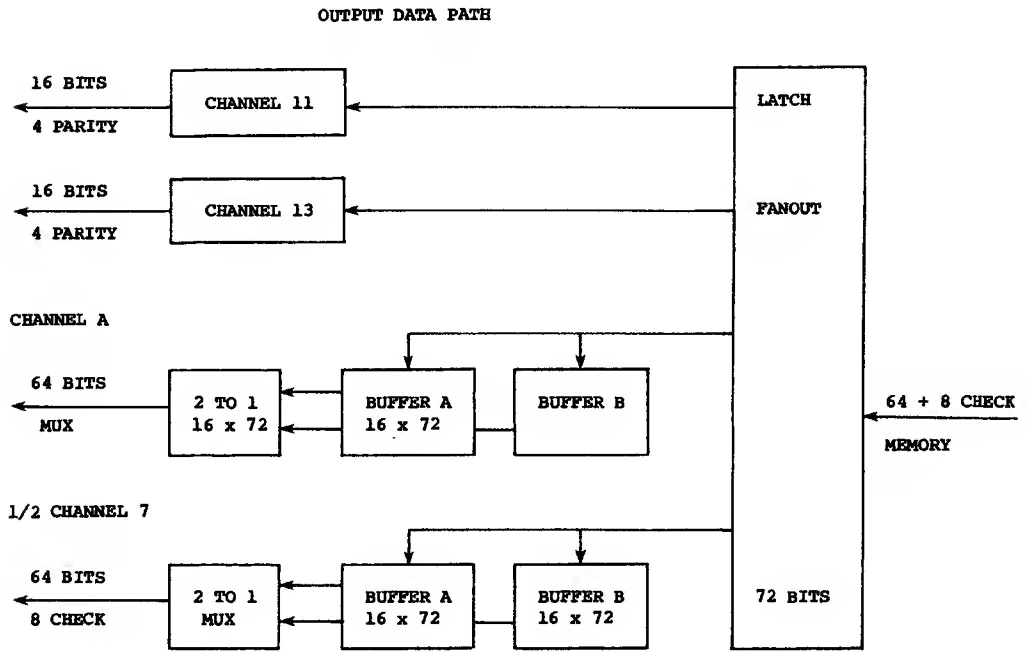


Figure 2-8. Input/output data paths (for CPU 0)

I/O MEMORY REQUEST CONDITIONS

The following conditions must be present for an I/O memory request to be processed:

- I/O request
- Bank not busy
- No simultaneous conflicts with other memory ports
- No fetch request within the CPU pair
- No exchange sequence within the CPU pair

I/O MEMORY ADDRESSING

All I/O memory references are absolute. The CA and CL registers are 24 bits, allowing I/O access to all of memory. Setting of the CA and CL registers is limited to monitor mode. I/O memory reference addresses are not checked for range errors.

CPU CONTROL SECTION 3

INTRODUCTION

All CPUs have identical, independent control sections containing registers and instruction buffers for instruction issue and control. A control section uses an exchange mechanism for switching instruction execution from program to program. These registers and buffers and the exchange mechanism are described in this section. Memory field protection, programmable clock, and deadstart sequence are also described.

INSTRUCTION ISSUE AND CONTROL

The registers and instruction buffers involved with instruction issue and control are described in the following paragraphs. Figure 3-1 illustrates the general flow of instruction parcels through the registers and buffers.

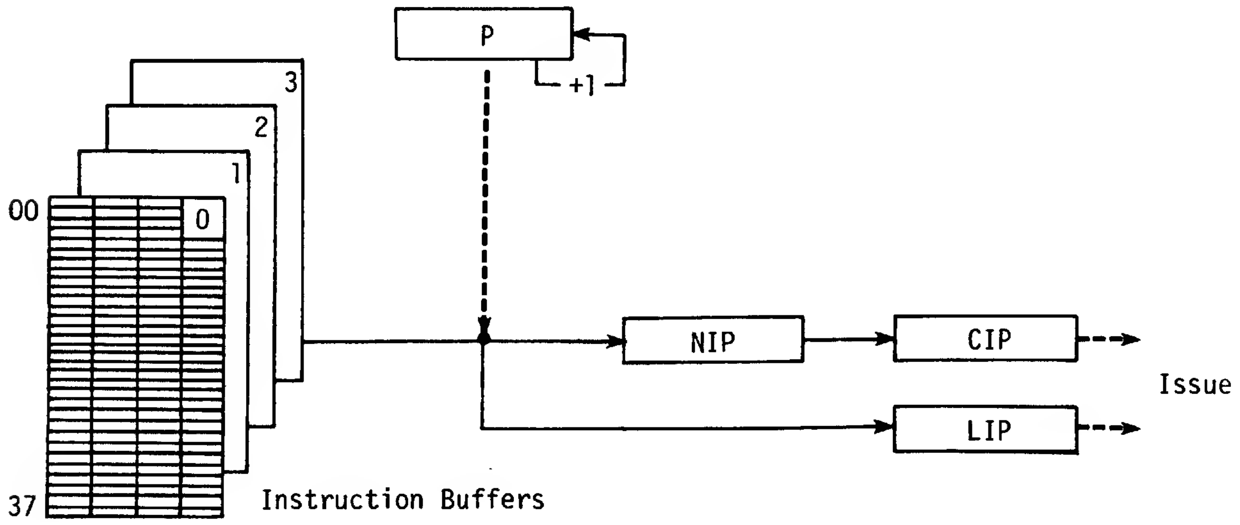


Figure 3-1. Instruction issue and control elements

PROGRAM ADDRESS REGISTER

The 24-bit Program Address (P) register indicates the next parcel of program code to enter the Next Instruction Parcel (NIP) register. The high-order 22 bits of the P register indicate the word address for the program word in memory relative to the base address. (Program size is limited to 4 million words.) The low-order 2 bits indicate the parcel within the word. Except on a branch instruction when the branch is taken or on an exchange, the contents of the P register are advanced 1 when an instruction parcel enters the NIP register.

New data enters the P register on an instruction branch or on an exchange sequence. (The exchange sequence is described under Exchange Mechanism later in this section.) The contents of P are then advanced sequentially until the next branch or exchange sequence. The value in the P register is stored directly into the terminating Exchange Package during an exchange sequence.

The P register is not master cleared. The value stored in P might not be accurate during the deadstart sequence.

NEXT INSTRUCTION PARCEL REGISTER

The 16-bit Next Instruction Parcel (NIP) register holds a parcel of program code before it enters the Current Instruction Parcel (CIP) register.

The NIP register is not master cleared. An undetermined instruction can issue during the master clear interval before the interrupt condition blocks data entry into the NIP register.

CURRENT INSTRUCTION PARCEL REGISTER

The 16-bit Current Instruction Parcel (CIP) register holds the instruction waiting to issue. The term *issue* indicates the transition of an instruction in CIP to its execution phase. If an instruction is a 2-parcel instruction, the CIP register holds the first parcel of the instruction and the Lower Instruction Parcel (LIP) register holds the second parcel. Issue of an instruction in CIP can be delayed until conflicting operations have been completed. Data arrives at the CIP register from the NIP register. Indicators making up the instruction are distributed to all modules having mode selection requirements when the instruction issues.

The control flags associated with the CIP register are master cleared; the register itself is not. An undetermined instruction can issue during the master clear sequence.

LOWER INSTRUCTION PARCEL REGISTER

The 16-bit Lower Instruction Parcel (LIP) register holds the second parcel of a 2-parcel instruction at the time the first parcel of the 2-parcel instruction is in the CIP register.

INSTRUCTION BUFFERS

A CPU has four instruction buffers, each can hold 128 consecutive 16-bit instruction parcels (figure 3-2). Instruction parcels are held in the buffers before being delivered to the NIP or LIP registers.

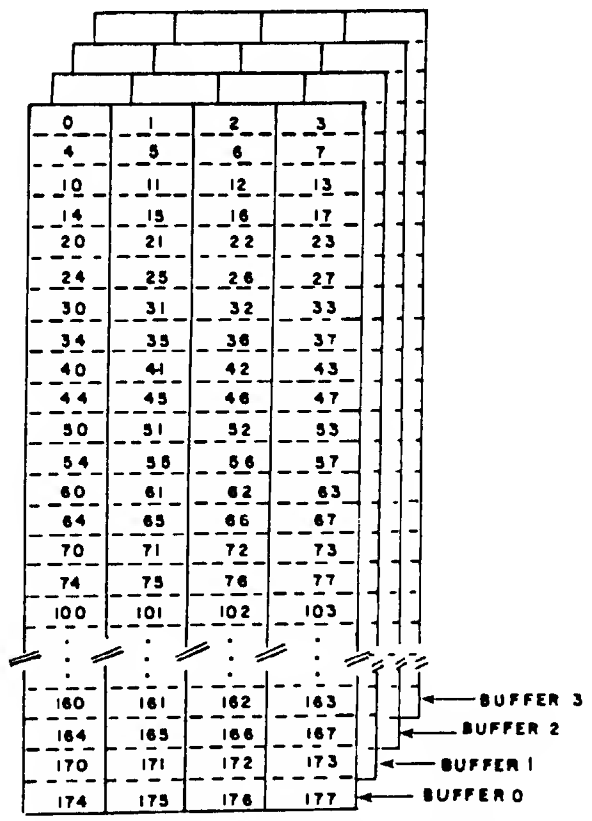


Figure 3-2. Instruction buffers

The beginning instruction parcel in a buffer always has a word address that is a multiple of 40_8 (a parcel address that is a multiple of 200_8) allowing the entire range of addresses for instructions in a buffer to be defined by the high-order 17 bits of the parcel address. Each buffer has a 17-bit Beginning Address register containing this value.

The Beginning Address registers are scanned each CP. If the high-order 17 bits of the P register match one of the beginning addresses, an in-buffer condition exists and the proper instruction parcel is selected from that instruction buffer. An instruction parcel to be executed normally is sent to the NIP. However, the second parcel of a 2-parcel instruction is blocked from entering the NIP register and is sent to the LIP register instead. The second parcel of the 2-parcel instruction becomes available when the first parcel issues from the CIP register. At the same time, an all-zero parcel is entered into the NIP register.

On an in-buffer condition, if the instruction is in a different buffer than the previous instruction, a change of buffers occurs requiring a 2-CP delay of the instruction reaching the NIP register.

An out-of-buffer condition exists when the high-order 17 bits of the P register do not match any instruction buffer beginning address. When this condition occurs, instructions must be loaded from memory into one of the instruction buffers before execution can continue. A 2-bit counter determines the instruction buffer receiving the instructions. Each out-of-buffer condition causes the counter to be incremented by 1 so that the buffers are selected in rotation.

Buffers are loaded from memory at the rate of eight words per CP. The first group of 32 parcels delivered to the buffer always contains the next instruction required for execution. For this reason, the branch out-of-buffer time is 16 CPs for 64-bank memories, providing memory is not busy (if busy, the branch fetch is delayed until the busy is resolved). Once the fetch proceeds, the remaining groups arrive at a rate of 32 parcels per CP and circularly fill the buffer.

An exchange sequence voids the instruction buffers, preventing a match with the P register and causing the buffers to be loaded as needed.

Forward and backward branching is possible within buffers. Branching does not cause reloading of an instruction buffer if the address of the instruction being branched to is within one of the buffers. Multiple copies of instruction parcels cannot occur in the instruction buffers. Because instructions are held in instruction buffers before issue and after (until the buffer is reloaded), self-modifying code should not be used. Also, because of independent data and instruction memory protection, self-modifying code may be impossible. As long as the address of the unmodified instruction is in an instruction buffer, the modified instruction in memory is not loaded into an instruction buffer.

Although optimizing code segment lengths for instruction buffers is not a prime consideration when programming a CPU, the number and size of the buffers and the capability for forward and backward branching can be used to good advantage. Large loops containing up to 512 consecutive instruction parcels can be maintained in the four buffers. An alternative is for a main program sequence in one or two of the buffers to make repeated calls to short subroutines maintained in the other buffers. The program and subroutines remain undisturbed in the buffers as long as no out-of-buffer condition or exchange causes reloading of a buffer.

EXCHANGE MECHANISM

A CPU uses an exchange mechanism for switching instruction execution from program to program. This exchange mechanism involves the use of blocks of program parameters known as Exchange Packages and a CPU operation referred to as an exchange sequence. For the convenience of Cray Assembly Language (CAL) programmers, an alternate bit position representation is used when discussing the Exchange Package. The bits are numbered from left to right with bit 0 assigned to the 2^{63} bit position.

EXCHANGE PACKAGE

The Exchange Package (figure 3-3) is a 16-word block of data in memory associated with a particular computer program. The Exchange Package contains the basic parameters necessary to provide continuity from one execution interval for the program to the next.

The Exchange Package contents (table 3-1) are arranged in a 16-word block. The exchange sequence swaps data from memory to the operating registers and back to memory. This sequence exchanges data in an active Exchange Package residing in the operating registers with an inactive Exchange Package in memory. The Exchange Address (XA) register address of the active Exchange Package specifies the memory address to be used for the swap. Data is exchanged and a new program execution interval is initiated by the exchange sequence.

The contents of the B, T, V, VM, SB, ST, and SM registers are not swapped in the exchange sequence. Data in these registers must be stored and replaced as required by specific coding in the program supervising the object program execution or by any program that needs this data. (See section 4 for descriptions of the operating registers and the VL register.)

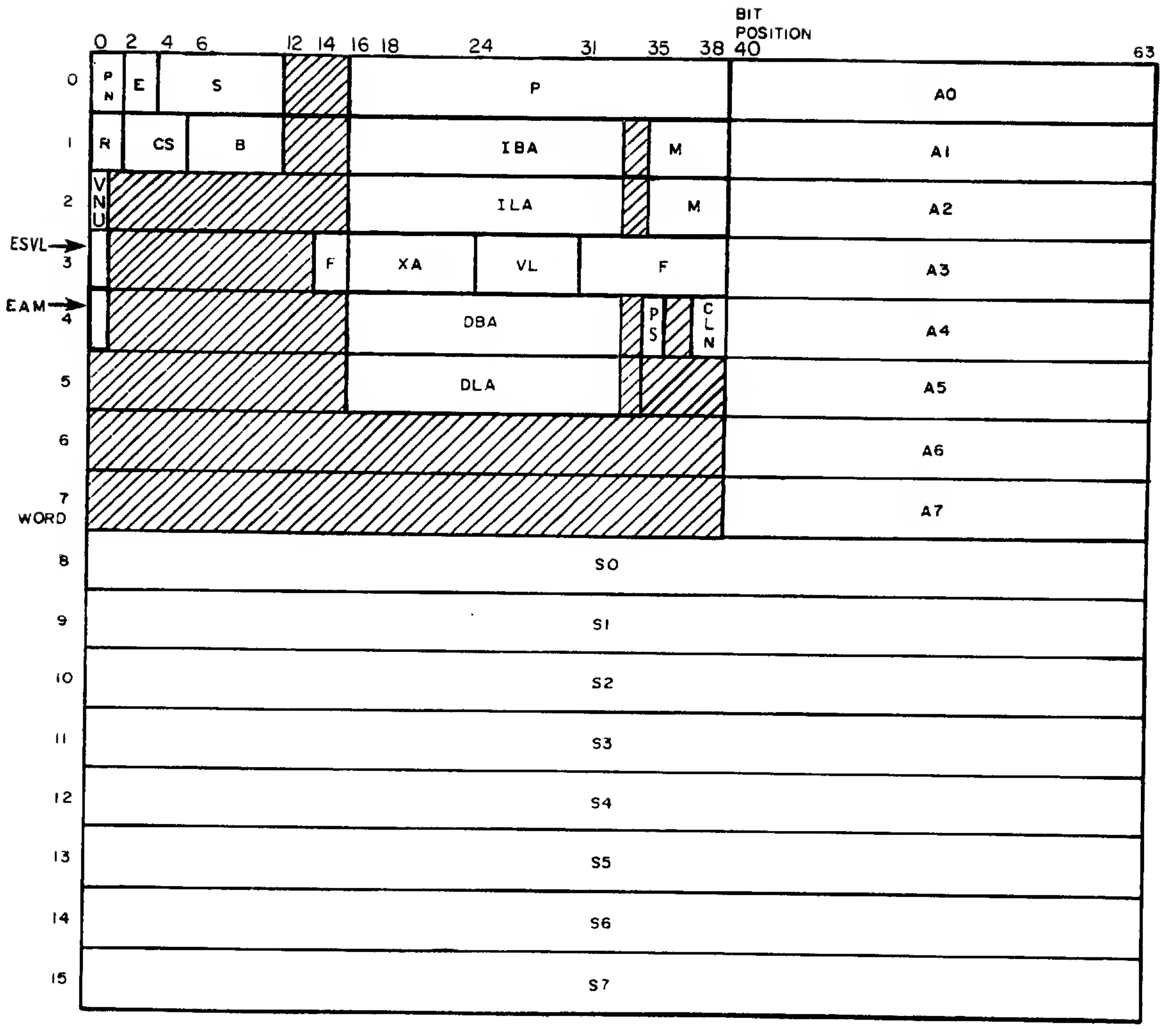


Figure 3-3. Exchange Package for a 4-processor system

Table 3-1. Exchange Package assignments

Field	Word	Bits
Processor number (PN)	0	0-1
Error type (E)	0	2-3
Syndrome bits (S)	0	4-11
Program Address register (P)	0	16-39
Read mode (R)	1	0-1
Read address (CSB)	1	2-5 (CS); 6-11 (B)
Instruction Base Address (IBA)	1	16-33
Instruction Limit Address (ILA)	2	16-33
Mode register (M)	1-2	35-39
Vector not used (VNU)	2	0
Enable Second Vector Logical (ESVL)	3	0
Flag register (F)	3	14-15; 31-39
Exchange Address register (XA)	3	16-23
Vector Length register (VL)	3	24-30
Enhanced Addressing Mode (EAM)	4	0
Data Base Address (DBA)	4	16-33
Program State (PS)	4	35
Cluster Number (CLN)	4	37-39
Data Limit Address (DLA)	5	16-33
Eight A register contents	0-7	40-63
Eight S register contents	8-15	0-63

Processor number

The contents of the 2-bit processor number (PN) position in the Exchange Package indicates in which CPU the Exchange Package executed. This value is not read into the CPU; it is a constant inserted only into a package being stored.

Vector not used (VNU)

The content of the vector not used (VNU) position in the Exchange Package indicates whether or not instructions 076, 077, or 140 through 177 where issued during the execution interval. If none of the instructions were issued, the bit remains set. If one or more of the instructions issued, the bit is cleared. Once cleared, the bit will remain clear until reset through a memory store to the dormant Exchange Package.

Enable Second Vector Logical (ESVL)

The content of the enable second vector logical (ESVL) position in the Exchange Package indicates whether or not the Second Vector Logical unit can be used. If set, instructions 140 through 145 may select the Second Vector Logical unit. If clear, the Second Vector Logical unit cannot be used; only the Full Vector Logical unit may be used.

Enhanced Addressing Mode (EAM)

The content of the enhanced addressing mode (EAM) position in the Exchange Package indicates whether or not address extension will take place for address calculations. If set, instructions 100 through 137 will sign-extend the 22-bit value (*jkm*) to 24 bits for address calculations (compatible with an 8-million word system). If clear, all CPU memory addresses (not I/O) will have address bits 2^{22} and 2^{23} replaced by data base address bits 2^{22} and 2^{23}, respectively.

Memory error data

Bit 36 (interrupt on correctable memory error bit) and bit 38 (interrupt on uncorrectable memory error bit) in the M (mode) register determine if memory error data is included in the Exchange Package. Error data, consisting of four fields of information, appears in the Exchange Package if bit 36 is set and correctable memory error is encountered or if bit 38 is set and an uncorrectable memory error is detected.†

Memory error data fields are described below.

E (Error type)	The type of memory error encountered, uncorrectable or correctable, is indicated in word 0, bits 2 and 3 of the Exchange Package. Bit 2 is set for an uncorrectable memory error; bit 3 is set for a correctable memory error.
S (Syndrome)	The 8 syndrome bits used in detecting a memory data error are returned in word 0, bits 4 through 11 of the Exchange Package. See section 2 for additional information.
R (Read mode)	This field indicates the read mode in progress when a memory data error occurred and is in word 1, bits 0 and 1 of the Exchange Package. These bits assume the following values:

00	I/O
01	Scalar (memory references with A or S)
10	Vector, B, or T
11	Instruction fetch or exchange

† For multiple bit memory errors, the hardware always sets the correctable Memory Error flag in the interrupted Exchange Package.

CSB (Read address)	The 10-bit CSB field contains the address where a memory data error occurred. Word 1, bits 6-11 (B) of the Exchange Package contain bits 2^5 through 2^0 of the address and can be considered the bank address. Word 1, bits 2 through 5 (CS) of the Exchange Package contain bits 2^{22} through 2^{19} (chip select) of the address.

EXCHANGE REGISTERS

Three special registers are instrumental in the exchange mechanism: the Exchange Address (XA) register, the Mode (M) register, and the Flag (F) register. These three registers are described below.

Exchange Address register

The 8-bit Exchange Address (XA) register specifies the first word address of a 16-word Exchange Package loaded by an exchange operation. The register contains the high-order 8 bits of a 12-bit field specifying the address. The low-order bits of the field are always 0; an Exchange Package must begin on a 16-word boundary. The 12-bit limit requires that the absolute address be in the lower 4096 ($10,000_8$) words of memory.

When an execution interval terminates, the exchange sequence exchanges the contents of the registers with the contents of the Exchange Package at the beginning address (XA) in memory.

Mode register

The 10-bit Mode (M) register contains part of the Exchange Package for a currently active program. The M register bits are assigned in words 1 and 2 of the Exchange Package as follows.

Word 1

Bit	Description
35	Waiting for Semaphore (WS) flag; when set, the CPU exchanged when a test and set instruction was holding in the CIP register.
36	Floating-point Error Status (FPS) flag; when set, a floating-point error has occurred regardless of the state of the Floating-point Error Mode flag.
37	Bidirectional Memory Mode (BDM) flag; when set, block reads and writes can operate concurrently.

Word 1 (continued)

Bit	Description
38	Selected for External Interrupts (SEI) flag; when set, this CPU is preferred for I/O interrupts.
39	Interrupt Monitor Mode (IMM) flag; when set, enables all interrupts in monitor mode except PC, MCU, I/O, and normal exit.

Word 2

Bit	Description
35	Operand Range Error Mode (IOR) flag; when set, enables interrupts on operand range errors.
36	Correctable Memory Error Mode (ICM) flag; when set, enables interrupts on correctable memory data errors.
37	Floating-point Error Mode (IFP) flag; when set, enables interrupts on floating-point errors.
38	Uncorrectable Memory Error Mode (IUM) flag; when set, enables interrupts on uncorrectable memory data errors.
39	Monitor Mode (MM) flag; when set, inhibits all interrupts except memory errors.

The 10 bits are set selectively during an exchange sequence.

Word 1, bit 37 (Bidirectional Memory Mode flag) can be set or cleared by using instructions 0026 (enable bidirectional Memory transfers) and 0025 (disable bidirectional Memory transfers).

Word 2, bit 35 (Operand Range Error Mode flag) can be set or cleared during the execution interval of a program by using instructions 0023 (enable interrupt on operand range error) and 0024 (disable interrupt on operand range error).

Word 2, bit 37 (Floating-point Error Mode flag) can be set or cleared during the execution interval for a program by using instructions 0021 (enable interrupt on floating-point error) and 0022 (disable interrupt on floating-point error).

Word 1, bits 36 and 37 and word 2, bits 35 and 37 can be read with instruction 073*i*01. Word 1, bits 35 and 36 indicate the state of the CPU at the time of the exchange. The remaining bits are not altered during the execution interval for the Exchange Package and can be altered only when the Exchange Package is inactive in storage.

Flag register

The 11-bit Flag (F) register contains part of the Exchange Package for the currently active program. This register is located in word 3 and contains 11 flags individually identified within the Exchange Package. Setting any of these flags interrupts program execution. When one or more flags are set, a Request Interrupt signal is sent to initiate an exchange sequence. The contents of the F register are stored along with the rest of the Exchange Package. The monitor program can analyze the 11 flags for the cause of the interruption. Before the monitor program exchanges back to the package, it must clear the flags in the F register area of the package. If any bit remains set, another exchange occurs immediately.

The F register bits are assigned in word 3 of the Exchange Package as follows.

Word 3

Bit	Description
14	Interrupt From Internal CPU (ICP) flag; set when the another CPU issues instruction 001401.
15	Deadlock (DL) flag; set when all CPUs in a cluster are holding issue on a test and set instruction.
31	Programmable Clock Interrupt (PCI) flag; set when the interrupt countdown counter in the programmable clock equals 0. The programmable clock is explained later in this section.
32	MCU Interrupt (MCU) flag; set when the MIOP sends this signal.
33	Floating-point Error (FPE) flag; set when a floating-point range error occurs in any of the floating-point functional units and the Enable Floating-point Interrupt flag is set. Floating-point functional units are explained in section 4, Computation.
34	Operand Range Error (ORE) flag; set when a data reference is made outside the boundaries of the Data Base Address (DBA) and Data Limit Address (DLA) registers and the Enable Operand Range Interrupt flag is set. Operand range error is explained later in this section.
35	Program Range Error (PRE) flag; set when an instruction fetch is made outside the boundaries of the Instruction Base Address (IBA) and Instruction Limit Address (ILA) registers. Program range error is explained later in this section.

Word 3 (continued)

Bit	Description
36	Memory Error (ME) flag; set when a correctable or uncorrectable memory error occurs and the corresponding enable memory error mode bit is set in the M register.
37	I/O Interrupt (IOI) flag; set when a 6 Mbyte channel or the 1250 Mbyte channel completes a transfer.
38	Error Exit (EEX) flag; set by an error exit instruction (000).
39	Normal Exit (NEX) flag; set by a normal exit instruction (004).

Any flag (except the Memory Error flag) can be set in the F register only if the active Exchange Package is not in monitor mode. Such flags are set only if word 2, bit 39 of the M register is 0. Except for the Memory Error flag, if the program is in monitor mode and the conditions for setting an F register are present, the flag remains cleared and no exchange sequence is initiated.

Cluster Number register

The 3-bit Cluster Number (CLN) register determines the CPU's cluster. The contents of the CLN register are used to determine which set of SB, ST, and SM registers the CPU can access. If the CLN register is 0, then the CPU does not have access to SB, ST, or SM registers. The contents of the CLN registers in all CPUs are also used to determine the condition necessary for a deadlock interrupt.

Program State register

The content of the 1-bit Program State (PS) register is manipulated by the operating system to represent different program states in the CPUs concurrently processing a single program.

A registers

The current contents of all A registers are stored in bits 40 through 63 of words 0 through 7 during exchange.

S registers

The current contents of all S registers are stored in bits 0 through 63 of words 8 through 15 during exchange.

Program Address register

The contents of the Program Address (P) register (address of first program instruction not yet issued) are stored in bits 16 through 39 of word 0 (maximum program size is 4 million words). The instruction at this location is the first instruction to be issued when this program begins again.

Memory field registers

Each object program has a designated field of memory for instructions and data that is specified by the monitor program when the object program is loaded and initiated. All memory addresses contained in the object program code are relative to one of two base addresses specifying the beginning of the appropriate field, and limited in size. Each object program reference to memory is checked against the limit and base addresses to determine if the address is within the bounds assigned. These field limits are contained in four registers that are saved in the Exchange Package. The four registers are: the Instruction Base Address (IBA) register, the Instruction Limit Address (ILA) register, the Data Base Address (DBA) register, and the Data Limit Address (DLA) register. Refer to the subsection on Memory Field Protection later in this section for an explanation of the registers.

ACTIVE EXCHANGE PACKAGE

An active Exchange Package resides in the operating registers. The interval of time when the Exchange Package and the program associated with it are active is called the execution interval. An execution interval begins with an exchange sequence where the subject Exchange Package moves from memory to the operating registers. An execution interval ends as the Exchange Package moves back to memory in a subsequent exchange sequence.

EXCHANGE SEQUENCE

The exchange sequence is the vehicle for moving an inactive Exchange Package from memory into the operating registers. At the same time, the exchange sequence moves the currently active Exchange Package from the operating registers back into memory. This swapping operation is done in a fixed sequence when all computational activity associated with the currently active Exchange Package has stopped. The same 16-word block of memory is used as the source of the inactive Exchange Package and the destination of the currently active Exchange Package. Location of this block is specified by the content of the XA register and is a part of the currently active Exchange Package. The exchange sequence can be initiated by deadstart sequence, Interrupt flag set, or program exit.

Exchange initiated by deadstart sequence

The deadstart sequence forces the XA register content to 0 for all CPUs and also forces an interrupt in one CPU. These two actions cause an exchange using memory address 0 as the location of the Exchange Package. The inactive Exchange Package at address 0 then moves into the operating registers and initiates a program using these parameters. The Exchange Package swapped to address 0 is largely indeterminate because of the deadstart operation. New data entered at these storage addresses then discards the old Exchange Package in preparation for starting subsequent CPUs with an interprocessor interrupt.

When instruction 0014*j*1 (IP) is issued in the first CPU, the CPU associated with processor number *j* exchanges to address 0 in memory. (A set of switches on the mainframe's control panel associates processor number with CPU number and selects which CPU is deadstarted first.)

Exchange initiated by Interrupt flag set

An exchange sequence can be initiated by setting any one of the Interrupt flags in the F register. Setting of one or more flags causes a Request Interrupt signal to initiate an exchange sequence.

Exchange initiated by program exit

Two program exit instructions initiate an exchange sequence. Timing of the instruction execution is the same in either case, the difference is determined by which of the two flags is set in the F register. The two instructions are:

000	ERR	Error exit
004	EX	Normal exit

The two exits enable a program to request its own termination. A non-monitor (object) program usually uses the normal exit instruction to exchange back to the monitor program. The error exit allows for abnormal termination of an object program. The exchange address selected is the same as for a normal exit.

Each instruction has a flag in the F register. The appropriate flag is set if the currently active Exchange Package is not in monitor mode. The inactive Exchange Package called in this case is normally one that executes in monitor mode. Flags are checked for evaluation of the program termination cause.

The monitor program selects an inactive Exchange Package for activation by setting the address of the inactive Exchange Package in the XA register and then executing a normal exit instruction.

Exchange sequence issue conditions

The following are hold issue conditions, execution time, and special cases for an exchange sequence.

Hold conditions:

- NIP register contains a valid instruction
- S, V, or A registers busy

Execution time:

For 64 banks, 40 CPs; consists of an exchange sequence (24 CPs) and a fetch operation (16 CPs).

Special cases:

If a test and set instruction is holding in the CIP register, both CIP and NIP registers are cleared and the exchange occurs with the WS (Waiting for Semaphore) flag set and the P register pointing to the test and set instruction.

EXCHANGE PACKAGE MANAGEMENT

Each 16-word Exchange Package resides in an area defined during system deadstart. The defined area must lie within the lower 4096 ($10,000_8$) words of memory. The package at address 0 is the deadstart monitor program's Exchange Package. Other packages provide for object programs and monitor tasks. Non-monitor packages lie outside of the field lengths for the programs they represent as determined by the base and limit addresses for the programs. Only the monitor program has a field defined so that it can access all of memory, including Exchange Package areas. The defined field allows the monitor program to define or alter all Exchange Packages other than its own when it is the currently active Exchange Package. Since no interlock exists between an exchange sequence in a CPU and memory transfers in another CPU, modification of Exchange Packages which can be used by another CPU should be avoided, except under software controlled situations.

Proper management of Exchange Packages dictates that a non-monitor program always exchanges back to the monitor program that exchanged to it. The exchange ensures that the program information is always exchanged into its proper Exchange Package.

For example, the monitor program (A) begins an execution interval following deadstart. No interrupts (except memory) can terminate its execution interval since it is in monitor mode. Program A voluntarily exits by issuing a normal exit instruction (004). However, before doing so, program A sets the contents of the XA register to point to the user program (B) Exchange Package so that program B is the next program to execute. Program A sets the exchange address in program B's Exchange Package to point back to program A.

The exchange sequence to program B causes the exchange address from program B's Exchange Package to be entered in the XA register. At the same time, the exchange address in the XA register goes to program B's Exchange Package area with all other program parameters for program A. When the exchange is complete, program B begins its execution interval.

To illustrate the exchange sequence, assume that while program B is executing, an Interrupt flag sets initiating an exchange sequence. Since program B cannot alter the XA register, the exit is back to program A. Program B's parameters exchange back into its Exchange Package area; program A's parameters held in program B's package area during the execution interval exchange back into the operating registers.

Program A, upon resuming execution, determines an interrupt has caused the exchange and sets the XA register to call the proper interrupt processor into execution. To do this, program A sets XA to point to the Exchange Package for the interrupt processing program (C). Program A clears the interrupt and initiates execution of program C by executing a normal exit instruction (004). Depending on the operating task, program C can execute in monitor mode or in user mode.

Further information on Exchange Package management is contained in the COS EXEC/STP/CSP Internal Reference Manual, publication SM-0040.

MEMORY FIELD PROTECTION

At execution time, each object program has a designated field of memory for instructions and data. The field limits are specified by the monitor program when the object program is loaded and initiated. The fields can begin at any word address that is a multiple of 64 (that is, 100_8) and can continue to another address that is one less than a multiple of 64. The fields can overlap.

All memory addresses contained in the object program code are relative to one of the two base addresses specifying the beginning of the appropriate field. An object program cannot read or alter any memory location with an absolute address lower than that base address. Each object program reference to memory is checked against the limit and base addresses to determine if the address is within the bounds assigned. A memory read reference beyond the assigned field limits issues and completes, but a zero value is transferred from memory. A memory write reference beyond the assigned field limits is allowed to issue, but no write occurs.

Field limits are contained in four registers: the Instruction Base Address (IBA) register, the Instruction Limit Address (ILA) register, the Data Base Address (DBA) register, and the Data Limit Address (DLA) register. These four registers and flags associated with the field limits are described below.

INSTRUCTION BASE ADDRESS REGISTER

The Instruction Base Address (IBA) register holds the base address of the user's instruction field. An instruction can only be executed by the CPU if the absolute address at which the instruction is located is greater than or equal to the contents of the current Exchange Package IBA register of the program executing. This determination is made at instruction buffer fetch time by the CPU.

The contents of the IBA register are interpreted as the high-order 18 bits of a 24-bit memory address. The low-order 6 bits of the address are assumed to be 0 because of the number of banks, 64 decimal banks. Absolute memory addresses for an instruction fetch are formed by adding the IBA register to the P register (high-order 22 bits) modulo two to the twenty-fourth power.

A reference to an absolute address less than the address defined by IBA can only occur through a jump or branch instruction to an address beyond the memory capacity of the machine.

INSTRUCTION LIMIT ADDRESS REGISTER

The Instruction Limit Address (ILA) register holds the limit address of the user's field. An instruction can only be executed by the CPU if the absolute address where it is located is less than the contents of the current Exchange Package ILA register of the program executing. This determination is made at instruction buffer fetch time by the CPU.

The contents of the ILA register are interpreted as the high-order 18 bits of a 24-bit memory address. The low-order 6 bits of the address are assumed to be 0 because of the number of banks, 64 (decimal) banks. The largest absolute address that can be executed by a program is defined by $[(\text{ILA}) \times 2^6] - 1$.

If the final absolute address of the instruction buffer fetch as computed by the CPU does not fall between the range of addresses contained within the currently executing Exchange Package IBA and ILA registers, the CPU generates a program range error interrupt.

DATA BASE ADDRESS REGISTER

The Data Base Address (DBA) register holds the base address of the user's data field. An operand can only be fetched or stored by the CPU if the absolute address where the operand is located is greater than or equal to the contents of the current Exchange Package DBA register of the program executing. This determination is made each time an operand is fetched or stored by the CPU.

The contents of the DBA register are interpreted as the high-order 18 bits of a 24-bit memory address. The low-order 6 bits of the DBA register are assumed to be 0. Absolute memory addresses for operands are formed by adding the DBA register to the modified operand address modulo two to the twenty-fourth power.

DATA LIMIT ADDRESS REGISTER

The Data Limit Address (DLA) register holds the (upper) limit address of the user's data field. An operand can only be fetched or stored by the CPU if the absolute address where the operand is located is less than the contents of the current Exchange Package DLA register of the program executing. This determination is made each time an operand is fetched or stored by the CPU.

The contents of the DBA register are interpreted as the high-order 18 bits of a 24-bit memory address. The low-order 6 bits of the DBA register are assumed to be 0. The largest absolute address that can be referenced for data by a program is defined by $[(DLA) \times 2^6] - 1$.

If the final absolute address of the operand as computed by the CPU does not fall between the range of addresses contained within the currently executing Exchange Package DBA and DLA registers, the CPU generates an operand (address) range error interrupt.

PROGRAM RANGE ERROR

The Program Range Error flag sets if a memory reference outside the boundaries of the IBA and ILA registers is for an instruction fetch. An out-of-range memory reference can occur in a non-monitor mode program on a branch or jump instruction calling for a program address above or below the limits. The Program Range Error flag causes an error condition that terminates program execution. The monitor program checks the state of the Program Range Error flag and takes appropriate action, perhaps aborting the user program.

OPERAND RANGE ERROR

The Operand Range Error flag sets if the Operand Range Error Mode flag is set and a memory reference outside the boundaries of the DBA and DLA registers is called to read or write an operand for an A, B, S, T, or V register and the Operand Range Interrupt Error flag is set. The Operand Range Error flag causes an error condition that terminates the user program execution. The monitor program checks the state of the Operand Range Error flag and takes appropriate action, perhaps aborting the user program.

PROGRAMMABLE CLOCK

The programmable clock can be used to accurately measure the duration of intervals. Intervals selected under monitor program control generate a periodic interrupt. The clock frequency is 105 Mhz. Intervals from 9.5 nanoseconds to approximately 40.8 seconds are possible. Intervals shorter than 100 microseconds are not practical due to the monitor overhead involved in processing the interrupt. Supporting the programmable clock are the Interrupt Interval (II) register, the Interrupt Countdown (ICD) counter, and four monitor mode instructions.

INSTRUCTIONS

Four monitor mode instructions support the programmable clock:

0014*j*4	PCI S*j*	Enter Interrupt Interval (II) register with (S*j*).
001405	CCI	Clear the programmable clock interrupt request.
001406	ECI	Enable the programmable clock interrupt request.
001407	DCI	Disable the programmable clock interrupt request.

INTERRUPT INTERVAL REGISTER

The 32-bit Interrupt Interval (II) register can be loaded with a binary value equal to the number of CPs that are to elapse between programmable clock interrupt requests. The interrupt interval is transferred from the low-order 32 bits of the S*j* register into the II register and the ICD counter when instruction 0014*j*4 is executed.

This value is held in the II register and is transferred to the ICD counter each time the counter reaches 0 and generates an interrupt request. The content of the II register is changed only by another instruction 0014*j*4.

INTERRUPT COUNTDOWN COUNTER

The 32-bit Interrupt Countdown (ICD) counter is preset to the contents of the II register when instruction 0014*j*4 is executed. This counter runs continuously but counts down, decrementing by 1 each CP until the content of the counter is 0. The ICD sets the programmable clock interrupt request and samples the interval value held in the II register. The ICD repeats the countdown to zero cycle, setting the programmable clock interrupt request at regular intervals determined by the interval value.

When the programmable clock interrupt request is set, it remains set until a clear programmable clock interrupt request is executed. A programmable clock interrupt request can be set only after the enable programmable clock interrupt request is executed. A programmable clock interrupt request causes an interrupt only when not in monitor mode. A request set in monitor mode is held until the system switches to user mode.

CLEAR PROGRAMMABLE CLOCK INTERRUPT REQUEST

Following a program interrupt interval, an active programmable clock interrupt request can be cleared by executing instruction 001405.

Following any deadstart, the monitor program should ensure the state of the programmable clock interrupt by issuing instructions 001405 and 001407.

PERFORMANCE MONITOR

The system contains a set of eight performance counters to track certain hardware related events that can be used to indicate relative performance. The events that can be tracked are the number of specific instructions issued, hold issue conditions, the number of fetches, references, etc., and are selected through instruction 0015*j*0. Refer to Appendix C for complete information on performance monitoring.

DEADSTART SEQUENCE

The deadstart sequence of operations starts a program running in the mainframe after power has been turned off and then turned on again or whenever the operating system is to be reinitialized in the mainframe. All registers in the machine, all control latches, and all words in memory should be considered invalid after power has been turned on. The following sequence of operations to begin the program is initiated by the I/O Subsystem.

1. Turn on Master Clear signal.
2. Turn on I/O Clear signal.
3. Turn off I/O Clear signal.
4. Load memory via I/O Subsystem.
5. Turn off Master Clear signal.

The Master Clear signal halts all internal computation and forces critical control latches to predetermined states. The I/O Clear signal clears the input Channel Address register of the MCU channel and activates the MCU input channel. All other input channels remain inactive. The I/O Subsystem then loads an initial Exchange Package and monitor program. The Exchange Package must be located at address 0 in memory. Turning off the Master Clear signal initiates the exchange sequence to read this package and to begin execution of the monitor program in CPU 0 (PN=0).

The other CPUs remain in a master-cleared state until instruction 0014*j*1 (IP) is issued in the CPU with PN=0. Then the CPU with PN=*j* exchanges to address 0 in memory.

Because the exchange of CPU 0 overwrites the contents of the inactive Exchange Package at address 0, CPU 0 must reinitialize the Exchange Package at address 0 before allowing other CPUs to start. (Any CPU can be started first by using a switch on the control panel.) Subsequent actions are dictated by the design of the operating system.

CPU COMPUTATION SECTION 4

INTRODUCTION

Each CPU contains an identical, independent computation section. A computation section consists of operating registers and functional units associated with three types of processing: address, scalar, and vector. Address processing operates on internal control information such as addresses and indexes and has two levels of 24-bit registers and two integer arithmetic functional units. Scalar and vector processing are performed on data.

A vector is an ordered set of elements. A vector instruction operates on a series of elements repeating the same function and producing a series of results. Scalar processing starts an instruction, handles one operand or operand pair, then produces a single result.

The main advantage of vector over scalar processing is eliminating instruction start-up time for all but the first operand. Scalar processing has two levels of 64-bit scalar registers, four functional units dedicated solely to scalar processing, and three floating-point functional units shared with vector operations. Vector processing has a set of 64-element registers of 64 bits each, five functional units dedicated solely to vector applications, and three floating-point functional units supporting both scalar and vector operations.

Address information flows from Central Memory or from control registers to address registers. Information in the address registers is distributed to various parts of the control network for use in controlling the scalar, vector, and I/O operations. The address registers can also supply operands to two integer functional units. The units generate address and index information and return the result to the address registers. Address information can also be transmitted to Central Memory from the address registers.

Data flow in a computation section is from Central Memory to registers and from registers to functional units. Results flow from functional units to registers and from registers to Central Memory or back to functional units. Data flows along either the scalar or vector path depending on the processing mode. An exception is that scalar registers can provide one required operand for vector operations performed in the vector functional units.

Integer or floating-point arithmetic operations are performed in the computation section. Integer arithmetic is performed in twos complement mode. Floating-point quantities have signed magnitude representation.

Floating-point instructions provide for addition, subtraction, multiplication, and reciprocal approximation. The reciprocal approximation instructions provide for a floating-point divide operation using a multiple instruction sequence. These instructions produce 64-bit results (1-bit sign, 15-bit exponent, and 48-bit normalized coefficient).

Integer or fixed-point operations are integer addition, integer subtraction, and integer multiplication. Integer addition and subtraction operations produce either 24-bit or 64-bit results. An integer multiply operation produces a 24-bit result. A 64-bit integer multiply operation is done through a software algorithm using the floating-point multiply functional unit to generate multiple partial products. These partial products are then shifted and merged to form the full 64-bit product. No integer divide instruction is provided; the operation is accomplished through a software algorithm using floating-point hardware.

The instruction set includes Boolean operations for OR, AND, equivalence, and exclusive OR and for a mask-controlled merge operation. Shift operations allow the manipulation of either 64-bit or 128-bit operands to produce 64-bit results. With the exception of 24-bit integer arithmetic, most operations are implemented in vector and scalar instructions. The integer product is a scalar instruction designed for index calculation. Full indexing capability allows the programmer to index throughout memory in either scalar or vector modes. The index can be positive or negative in either mode. Indexing allows matrix operations in vector mode to be performed on rows or the diagonal as well as conventional column-oriented operations.

Population and parity counts are provided for both vector and scalar operations. An additional scalar operation is the leading zero count.

Characteristics of a CPU computation section are summarized below.

- Integer and floating-point arithmetic
- Twos complement integer arithmetic
- Signed magnitude floating-point arithmetic
- Address, scalar, and vector processing modes
- Fourteen functional units
- Eight 24-bit address (A) registers
- Sixty-four 24-bit intermediate address (B) registers
- Eight 64-bit scalar (S) registers
- Sixty-four 64-bit intermediate scalar (T) registers
- Eight 64-element vector (V) registers, 64 bits per element

OPERATING REGISTERS

Operating registers, a primary programmable resource of a CPU, enhance the speed of the system by satisfying heavy demands for data made by the functional units. A single functional unit can require one to three operands per clock period (CP) to perform the necessary functions and can deliver results at a rate of one per CP. Multiple functional units can be used concurrently.

A CPU has three primary and two intermediate sets of registers. The primary sets of registers are address, scalar, and vector, designated in this manual as A, S, and V, respectively. These registers are considered primary because functional units can access them directly.

For the A and S registers, an intermediate level of registers exists which is not accessible to the functional units but acts as a buffer for the primary registers. Block transfers are possible between these registers and Central Memory so that the number of memory reference instructions required for scalar and address operands is greatly reduced. The intermediate registers that support the A registers are referred to as B registers. The intermediate registers that support S registers are referred to as T registers.

ADDRESS REGISTERS

Figure 4-1 illustrates registers and functional units used for address processing. The two types of address registers are designated A registers and B registers and are described in the following paragraphs.

A REGISTERS

Eight 24-bit A registers serve a variety of applications but are primarily used as address registers for memory references and as index registers. They provide values for shift counts, loop control, and channel I/O operations and receive values of population count and leading zeros count. In address applications, A registers index the base address for scalar memory references and provide both a base address and an address increment for vector memory references.

The address functional units support address and index generation by performing 24-bit integer arithmetic on operands obtained from A registers and by delivering the results to A registers.

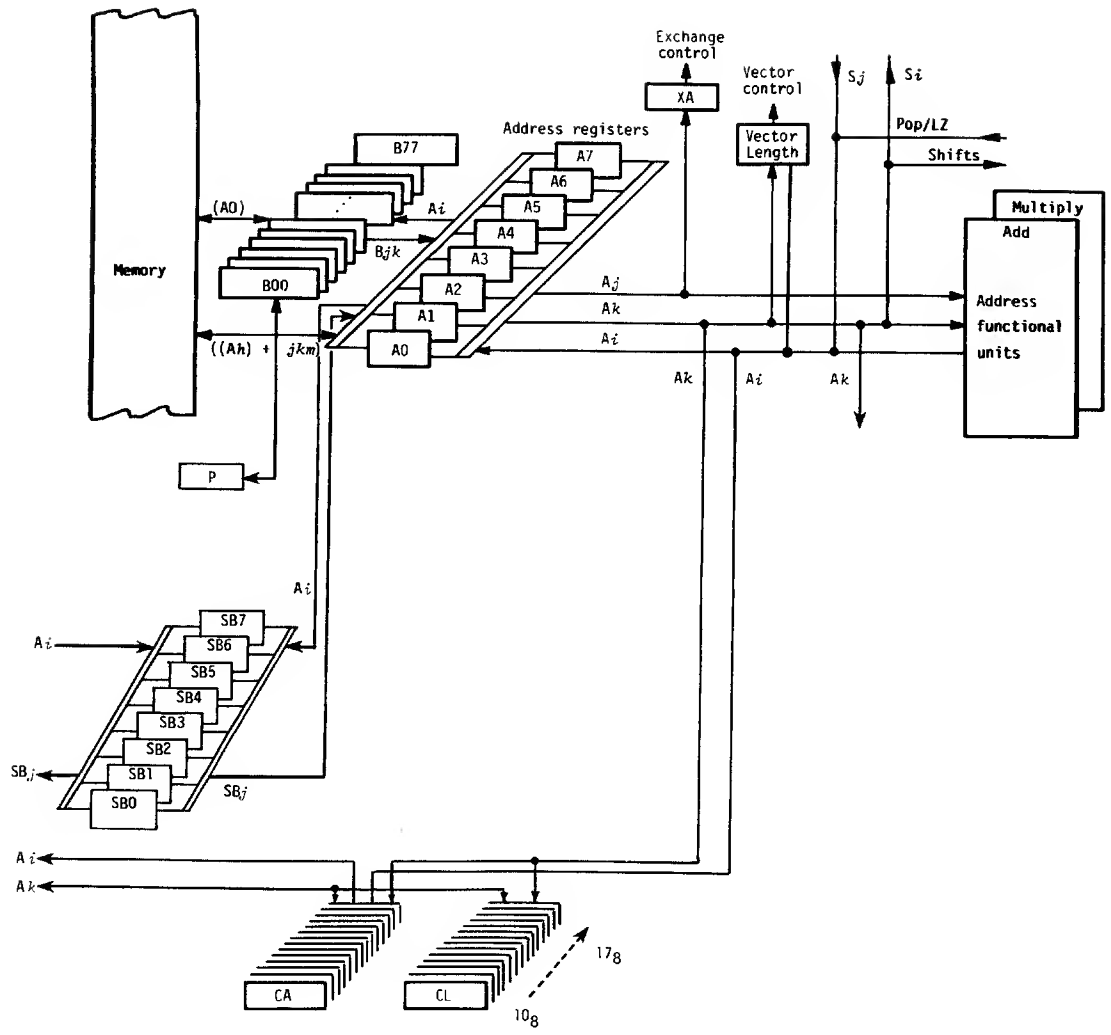


Figure 4-1. Address registers and functional units

Data is moved directly between Central Memory and A registers or is placed in B registers. Placing data in B registers allows buffering of the data between A registers and Central Memory. Data can also be transferred between A and S registers and between A and Shared Address (SB) registers.

The Vector Length (VL) register and Exchange Address (XA) register are set by transmitting a value to them from an A register. The VL register can also be transmitted to an A register. (The VL register is described under Vector Control Registers later in this section.)

When an issued instruction delivers new data to an A register, a reservation is set for that register. The reservation prevents issue of instructions that use the register until the new data is delivered.

In this manual, the A registers are individually referred to by the letter A followed by a number ranging from 0 through 7. Instructions reference A registers by specifying the register number as the *h*, *i*, *j*, or *k* designator as described in section 5.

The only register implicitly referenced is the A0 register as illustrated in the following instructions:

010*ijkm*	JAZ *exp*	Branch to *ijkm* if (A0)=0.
011*ijkm*	JAN *exp*	Branch to *ijkm* if (A0)≠0.
012*ijkm*	JAP *exp*	Branch to *ijkm* if (A0) is positive; includes (A0)=0.
013*ijkm*	JAM *exp*	Branch to *ijkm* if (A0) is negative.
034*ijk*	B*jk*,A*i* ,A0	Read (A*i*) words to B register *jk* from (A0).
035*ijk*	,A0 B*jk*,A*i*	Store (A*i*) words at B register *jk* to (A0).
036*ijk*	T*jk*,A*i* ,A0	Read (A*i*) words to T register *jk* from (A0).
037*ijk*	,A0 T*jk*,A*i*	Store (A*i*) words at T register *jk* to (A0).
176*i*0*k*	V*i* ,A0,A*k*	Read (VL) words to V*i* from (A0) incremented by (A*k*).
1770*jk*	,A0,A*k* V*j*	Store (VL) words from V*j* to (A0) incremented by (A*k*).

Section 5 of this manual contains additional information on the use of A registers by instructions.

B REGISTERS

A computation section contains sixty-four 24-bit B registers used as intermediate storage for the A registers. Typically, B registers contain data to be referenced repeatedly over a sufficiently long span, making it unnecessary to retain the data in either A registers or in Central Memory. Examples of uses are loop counts, variable array base addresses, and dimensions.

Transfer of a value between an A register and a B register requires only 1 CP. A block of B registers can be transferred to or from Central Memory at the maximum rate of one 24-bit value per CP. A reservation is made on all B registers during block transfers to and from B registers.

NOTE

Other instructions can issue on the CRAY X-MP while a block of B registers is being transferred to or from Central Memory.

In this manual, B registers are individually referred to by the letter B followed by a 2-digit octal number ranging from 00 through 77. Instructions reference B registers by specifying the B register number in the *jk* designator as described in section 5.

The only B register implicitly referenced is the B00 register. On execution of the return jump instruction, 007*ijkm*, register B00 is set to the next instruction parcel address (P) and a branch to an address specified by *ijkm* occurs. Upon receiving control, the called routine conventionally saves (B00) so that the B00 register is available for the called routine to initiate return jumps of its own. When a called routine wishes to return to its caller, it restores the saved address and executes instruction 0050*jk*. Conventionally, this instruction, which is a branch to (B*jk*), causes the address saved in B*jk* to be entered into the P register as the address of the next instruction parcel to be executed.

SCALAR REGISTERS

Figure 4-2 illustrates registers and functional units used for scalar processing. The two types of scalar registers are designated S registers and T registers and are described in the following paragraphs.

S REGISTERS

Eight 64-bit S registers are the principal scalar registers for a CPU serving as the source and destination for operands executing scalar arithmetic and logical instructions. Scalar functional units perform both integer and floating-point arithmetic operations.

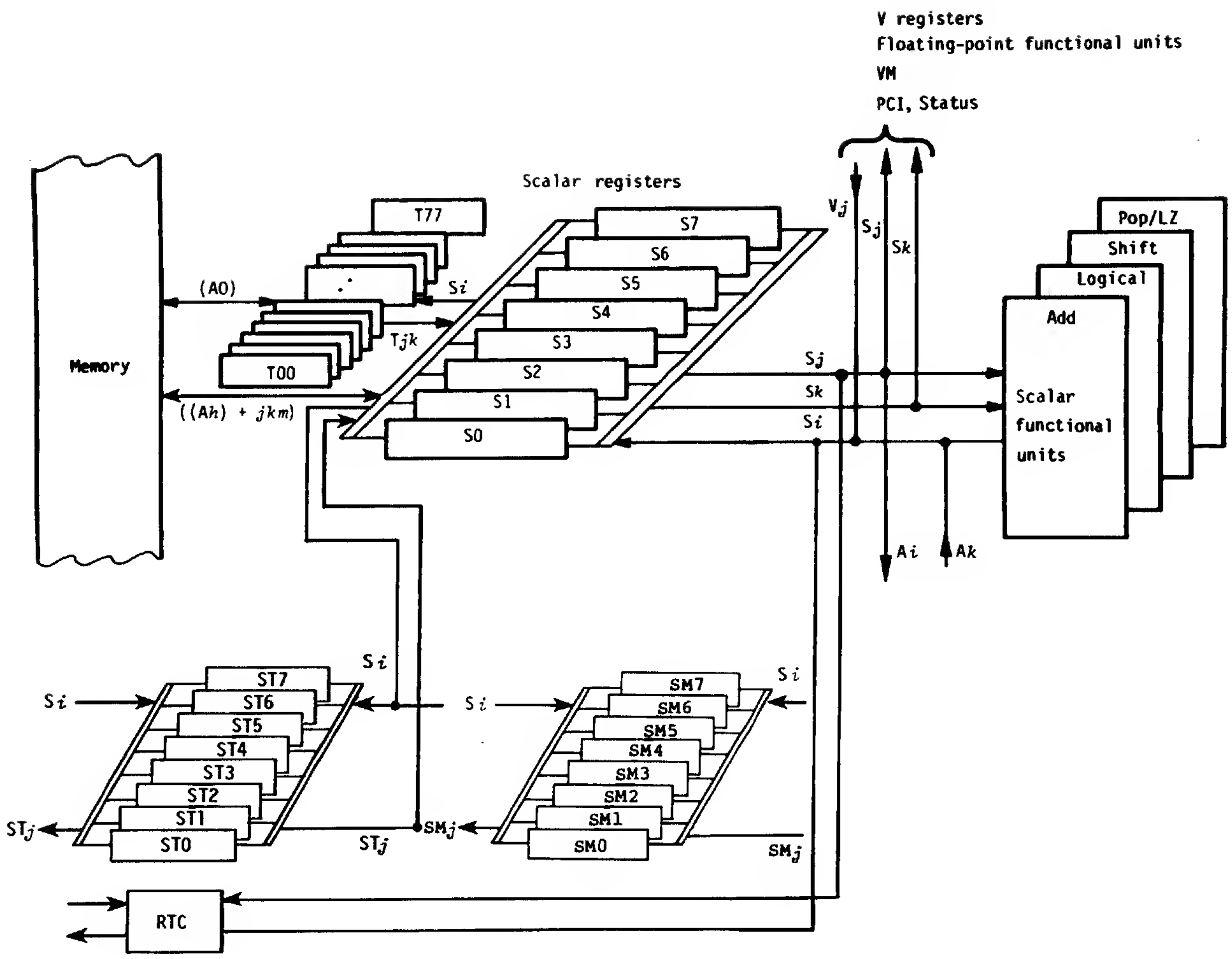


Figure 4-2. Scalar registers and functional units

S registers can furnish one operand in vector instructions. Single-word transmissions of data between an S register and an element of a V register are also possible.

Data is moved directly between Central Memory and S registers or is placed in T registers. This intermediate step allows buffering of scalar operands between S registers and Central Memory. Data is also transferred between A and S registers, between S and Shared Scalar (ST) registers, and between S and Semaphore (SM) registers.

Other uses of the S registers are the setting or reading of the Vector Mask (VM) register or the Real-time Clock (RTC) register or setting the Interrupt Interval (II) register.

When an issuing instruction delivers new data to an S register, a reservation is set for that register preventing issue of instructions that read the register until the new data is delivered.

In this manual, the S registers are individually referred to by the letter S followed by a number ranging from 0 through 7. Instructions reference S registers by specifying the register number as the *i*, *j*, or *k* designator as described in section 5.

The only register implicitly referenced is the S0 register as illustrated in the following instructions.

014*ijkm*	JSZ *exp*	Branch to *ijkm* if (S0)=0.
015*ijkm*	JSN *exp*	Branch to *ijkm* if (S0)≠0.
016*ijkm*	JSP *exp*	Branch to *ijkm* if (S0) is positive; includes (S0)=0.
017*ijkm*	JSM *exp*	Branch to *ijkm* if (S0) is negative.
052*ijk*	S0 S*i*<*exp*	Shift (S*i*) left *jk* places to S0.
053*ijk*	S0 S*i*>*exp*	Shift (S*i*) right *jk* places to S0.

The 8-bit Status register provides the status of the following flags:

- Processor Number (PN)
- Program State (PS)
- Cluster Number (CN)
- Floating-point Interrupts Enabled (IFP)
- Floating-point Error (FPE)
- Bidirectional Memory Enabled (BDM)
- Operand Range Interrupts Enabled (IOR)

Instruction 073 sends the contents of the Status register to an S register.

Section 5 of this manual has additional information on the use of S registers by instructions.

T REGISTERS

The computation section has sixty-four 64-bit T registers used as intermediate storage for the S registers. Data is transferred between T and S registers and between T registers and Central Memory. Transfer of a value between a T register and an S register requires only 1 CP. T registers reference Central Memory through block read and block write instructions. Block transfers occur at a maximum rate of one word per

CP. A reservation is made on all T registers during block transfers to and from T registers.

NOTE

Other instructions can issue on the CRAY X-MP while a block of T registers is being transferred to or from Central Memory.

In this manual, T registers are referred to by the letter T and a 2-digit octal number ranging from 00 through 77. Instructions reference T registers by specifying the octal number as the *jk* designator as described in section 5.

VECTOR REGISTERS

Figure 4-3 illustrates the registers and functional units used for vector operations. Vector registers and Vector Control registers are described in the following paragraphs.

V REGISTERS

The major computational registers of a CPU are eight V registers, each with 64 elements. Each V register element has 64 bits. When associated data is grouped into successive elements of a V register, the register quantity can be treated as a vector. Examples of vector quantities are rows or columns of a matrix or elements of a table. Computational efficiency is achieved by identically processing each element of a vector. Vector instructions provide for the iterative processing of successive V register elements. A vector operation always begins when operands are obtained from the first element of the operand V registers and the result is delivered to the first element of a V register. Successive elements are provided each CP and as each operation is performed, the result is delivered to successive elements of the result V register. The vector operation continues until the number of operations performed by the instruction equals a count specified by the content of the VL register.

Contents of a V register are transferred to or from Central Memory in a block mode by specifying a first word address in Central Memory, an increment or decrement for the Central Memory address or a set of indexes

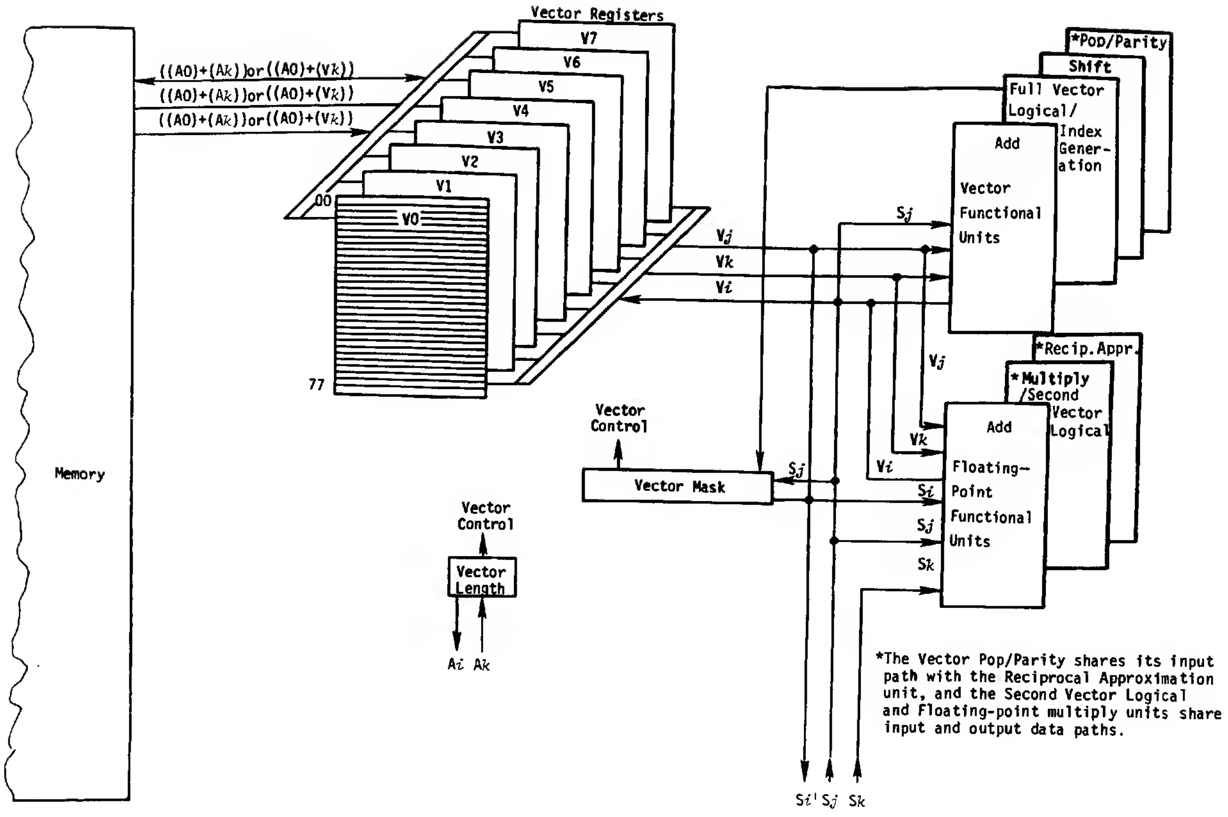


Figure 4-3. Vector registers and functional units

contained in a separate vector register, and a vector length. The transfer then proceeds beginning with the first element of the V register at a maximum rate of one word per CP, depending upon bank conflicts.

Discontinuities in the vector data stream can occur as a result of memory conflicts. These discontinuities, although not inhibiting chained operations, can appear in the chained operation data stream. Any discontinuity in the data stream adds proportionally to the total execution time of the vector operation.

Single-word data transfers are possible between an S register and an element of a V register.

Since many vectors exceed 64 elements, a long vector is processed as one or more 64-element segments and a possible remainder of less than 64 elements. Generally, it is convenient to compute the remainder and process this short segment before processing the remaining number of

64-element segments. However, a programmer can choose to construct the vector loop code in a number of ways. The processing of long vectors in FORTRAN is handled by the compiler and is transparent to the programmer.

A V register receiving results can also supply operands to a subsequent operation. Using a register as both a result and operand register in two different operations allows for the chaining together of two or more vector operations and two or more results can be produced per CP. Chained operations are detected automatically by a CPU and are not explicitly specified by the programmer. A programmer can reorder certain code segments to gain as much concurrency as possible in chained operations.

A conflict can occur between vector and scalar operations involving floating-point operations and memory access. With the exception of these operations, the functional units are always available for scalar operations. A vector operation occupies the selected functional unit until the vector is processed.

Parallel vector operations can be processed in two ways:

- Using different functional units and all different V registers

- Using the result stream from one V register simultaneously as the operand to another operation using a different functional unit (chain mode)

Parallel operations on vectors allow the generation of two or more results per CP. Most vector operations use two V registers as operands or one S and one V register as operands. Exceptions are vector shifts, vector logicals, vector reciprocals, and the load or store instructions.

In this manual, the V registers are individually referred to by the letter V followed by a number ranging from 0 through 7. Vector instructions reference V registers by specifying the register number as the *i*, *j*, or *k* designator as described in section 5.

Individual elements of a V register are designated in this manual by decimal numbers ranging from 00 through 63. These appear as subscripts to vector register references. For example, $V6_{29}$ refers to element 29 of V register 6.

NOTE

Parallel loading and storing of V registers is possible; two load operations and one store operation can occur simultaneously.

V register reservations and chaining

Reservation describes the condition of a register in use; that is, the register is not available for another operation as a result or as an operand register. Each register has two reservation conditions: one reserving it as a operand register and one reserving it as a result register. During execution of a vector instruction, reservations are placed on the operand V registers and on the result V register. These reservations are placed on the registers themselves, not on individual elements of the V register.

If a V register is reserved as a result and not as an operand, it can be used at any time as an operand and chaining occurs. This flexible chaining mechanism allows chaining to begin at any point in the result vector data stream. Full chaining occurs if the instruction causing chaining is issued before or at the time element 0 of the result arrives at the V register. Partial chaining occurs if the instruction issues after the arrival of element 0. Thus, the amount of concurrency in a chained operation depends upon the relationship between the issue time of the chaining instruction and the result vector data stream.

If a V register is reserved as an operand, it cannot be used as a result or operand register until the operand reservation clears. However, a V register can be used as both an operand and result in the same vector operation. A V register can serve only one vector operation as the source of one or both operands. A V register can serve only one vector operation as a result.

No reservation is placed on the VL register during vector processing. If a vector instruction employs an S register, no reservation is placed on the S register. The S register can be modified in the next instruction after vector issue without affecting the vector operation. The length and scalar operand (if appropriate) of each vector operation is maintained apart from the VL register and S register. Vector operations employing different lengths can proceed concurrently.

The A0 and A*k* registers in a vector memory reference are treated similarly and are available for modification immediately after use.

****

CAUTION

Cray Research, Inc., cautions against using a vector register as both a result and an operand if compatibility between a CRAY-1 and a CRAY X-MP system is necessary because vector recursion is not available on all Cray Research, Inc., computers.

****

VECTOR CONTROL REGISTERS

The Vector Length (VL) register and Vector Mask (VM) register provide control information needed in the performance of vector operations and are described below.

Vector Length register

The 7-bit Vector Length (VL) register is set to 1 through 100_8 (VL = 0 gives VL = 100_8) specifying the length of all vector operations performed by vector instructions and the length of the vectors held by the V registers. The VL register controls the number of operations performed for instructions 140 through 177 and is set to an A register value using instruction 0020 or read using instruction 023*i*01.

Vector Mask register

The Vector Mask (VM) register has 64 bits, each corresponding to a word element in a V register. Bit 2^{63} corresponds to element 0, bit 2^0 to element 63. The mask is used with vector merge and test instructions to allow operations to be performed on individual vector elements.

The VM register can be set from an S register through instruction 003 or can be created by testing a V register for a condition using instruction 175. The mask controls element selection in the vector merge instructions (146 and 147). Instruction 073 sends the contents of the VM register to an S register.

FUNCTIONAL UNITS

Instructions other than simple transmits or control operations are performed by specialized hardware known as functional units. Each unit implements an algorithm or a portion of the instruction set. Functional units have independent logic except for the Reciprocal Approximation, Vector Population Count, Floating-point Multiply and Second Vector Logical units (described later in this section), which share some logic. All functional units can be in operation at the same time.

A functional unit receives operands from registers and delivers the result to a register when the function has been performed. Functional units operate essentially in 3-address mode with source and destination addressing limited to register designators.

All functional units perform algorithms in a fixed amount of time; delays are impossible once the operands have been delivered to the unit. Time required from delivery of the operands to the functional unit until completion of the calculation is called the functional unit time and is measured in 9.5-nanosecond CPs.

Functional units are fully segmented. This means a new set of operands for unrelated computation can enter a functional unit each CP even though the functional unit time can be more than 1 CP. This segmentation is possible when information arrives at the functional unit and is held in the functional unit or moves within the functional unit at the end of every CP.

Fourteen functional units are identified in this manual and are arbitrarily described in four groups: address, scalar, vector, and floating-point. Each of the first three groups functions with one of the primary register types (A, S, and V) to support the address, scalar, and vector modes of processing available in the system. The fourth group, floating-point, supports either scalar or vector operations and accepts operands from or delivers results to S or V registers. In addition, Central Memory acts like a fifteenth functional unit for vector operations.

ADDRESS FUNCTIONAL UNITS

Address functional units perform 24-bit integer arithmetic on operands obtained from A registers and deliver the results to an A register. The arithmetic is twos complement.

Address Add functional unit

The Address Add functional unit performs 24-bit integer addition and subtraction. The unit executes instructions 030 and 031. Addition and subtraction are performed in a similar manner. The twos complement subtraction for instruction 031 occurs when the ones complement of the A*k* operand is added to the A*j* operand. Then a 1 is added in the low-order bit position of the result. No overflow is detected in the Address Add functional unit.

The Address Add functional unit time is 2 CPs.

Address Multiply functional unit

The Address Multiply functional unit executes instruction 032 forming a 24-bit integer product from two 24-bit operands. No rounding is performed. The result consists of the least significant 24 bits of the product.

This functional unit is designed to handle address manipulations not exceeding its data capabilities. The programmer must be careful when multiplying integers in the functional unit because the unit does not detect overflow of the product and significant portions of the product could be lost.

The Address Multiply functional unit time is 4 CPs.

SCALAR FUNCTIONAL UNITS

Scalar functional units perform operations on 64-bit operands obtained from S registers and, in most cases, deliver the 64-bit results to an S register. The exception is the Population/Leading Zero Count functional unit which delivers its 7-bit result to an A register.

Four functional units are exclusively associated with scalar operations and are described below. Three functional units are used for both scalar and vector operations and are described in the section on Floating-point Functional Units.

Scalar Add functional unit

The Scalar Add functional unit performs 64-bit integer addition and subtraction and executes instructions 060 and 061. Addition and subtraction are performed in a similar manner. The twos complement subtraction for instruction 061 occurs when the ones complement of the S*k* operand is added to the S*j* operand. Then a 1 is added in the low-order bit position of the result. No overflow is detected in the Scalar Add functional unit.

The Scalar Add functional unit time is 3 CPs.

Scalar Shift functional unit

The Scalar Shift functional unit shifts the entire 64-bit contents of an S register or shifts the double 128-bit contents of two concatenated S registers. Shift counts are obtained from an A register or from the *jk* portion of the instruction. Shifts are end off with zero fill. For a double shift, a circular shift is effected if the shift count does not exceed 64 and the *i* and *j* designators are equal and nonzero.

The Scalar Shift functional unit executes instructions 052 through 057. Single-shift instructions (052 through 055) have a functional unit time of 2 CPs. Double-shift instructions (056 and 057) have a functional unit time of 3 CPs.

Scalar Logical functional unit

The Scalar Logical functional unit performs bit-by-bit manipulation of 64-bit quantities obtained from S registers. It executes instructions 042 through 051, the mask, and Boolean instructions. Instructions 042 through 051 have a functional unit time of 1 CP.

Scalar Population/Parity/Leading Zero functional unit

This functional unit executes instructions 026 and 027. Instruction 026*ij*0 counts the number of bits in an S register having a value of 1 in the operand and has a functional unit time of 4 CPs. Instruction 026*ij*1 returns a 1-bit population parity count (even parity) of the S*j* register's contents. Instruction 027 counts the number of bits of 0 preceding a 1 bit in the operand and has a functional unit time of 3 CPs. For these instructions, the 64-bit operand is obtained from an S register and the 7-bit result is delivered to an A register.

VECTOR FUNCTIONAL UNITS

Most vector functional units perform operations on operands obtained from one or two V registers or from a V register and an S register. The Reciprocal, Shift, and Population/Parity functional units, which require only one operand, are exceptions. Results from a vector functional unit are delivered to a V register.

Successive operand pairs are transmitted each CP to a functional unit. The corresponding result emerges from the functional unit *n* CPs later, where *n* is the functional unit time and is constant for a given functional unit. The VL register determines the number of operand pairs to be processed by a functional unit.

The functional units described in this section are exclusively associated with vector operations. Three functional units are associated with both vector operations and scalar operations and are described in the subsection entitled Floating-point Functional Units. When a Floating-point functional unit is used for a vector operation, the general description of vector functional units given in the subsection applies.

Vector functional unit reservation

A functional unit engaged in a vector operation remains busy during each CP and cannot participate in other operations. In this state, the functional unit is reserved. Other instructions requiring the same functional unit will not issue until the previous operation is completed (with the exception of instructions 140 to 145, which may use either of the vector logical units). When the vector operation completes, the reservation is dropped and the functional unit is then available for another operation. A vector functional unit is reserved for (VL) + 4 CPs.

Vector Add functional unit

The Vector Add functional unit performs 64-bit integer addition and subtraction for a vector operation and delivers the results to elements of a V register. The unit executes instructions 154 through 157. Addition and subtraction are performed in a similar manner. For subtraction operations (156 and 157), the *Vk* operand is complemented before addition and a 1 is added into the low-order bit position of the result. No overflow is detected by the unit.

The Vector Add functional unit time is 3 CPs.

Vector Shift functional unit

The Vector Shift functional unit shifts the entire 64-bit contents of a V register element or the 128-bit value formed from two consecutive elements of a V register. Shift counts are obtained from an A register and are end off with zero fill.

All shift counts are considered positive unsigned integers. If any bit higher than 2^6 is set, the shifted result is all zeros.

The Vector Shift functional unit executes instructions 150 through 153. The functional unit time is 4 CPs for instruction 152, and the functional unit time is 3 CPs for instructions 150, 151, and 153.

Vector logical functional units

The CRAY X-MP Series model 48 has two vector logical functional units: a Full Vector Logical unit and a Second Vector Logical unit.

The Full Vector Logical unit performs bit-by-bit manipulations of the 64-bit quantities for instructions 140 through 147, logical operations associated with the vector mask instruction 175, and index generation. The Second Vector Logical unit performs bit-by-bit manipulations of 64-bit quantities for instructions 140 through 145 only.

Since both vector logical units can be used for instructions 140 through 145, when these instruction issues to the CIP register a selection is made to determine which vector functional unit will be used. Once a selection has been made, the instruction is committed to using that functional unit.

Normally, the instructions will attempt to issue first to the Second Vector Logical unit and then, if the unit is busy, attempt to issue to the Full Vector Logical unit. If both units are busy, the first unit to clear is selected. The Second Vector Logical unit may be busy because of another instruction or because the unit is disabled, see below. If there are other conflicts (register reservations) for the Second Vector Logical unit at the time the selection is made, the instructions will issue to the Full Vector Logical unit even though the Second Vector Logical unit clears before the instruction issues. When the Second Vector Logical unit is disabled, the functional unit busy always appears set and causes all 140 through 145 instructions to issue to the Full Vector Logical unit.

When the Second Vector Logical unit is enabled, it shares input and output data paths and the same functional unit busy with the Floating-point Multiply unit, so they cannot be used simultaneously. Also, since the Second Vector Logical unit ties up the Floating-point Multiply unit, some codes that rely on floating-point products may run slower if the Second Vector Logical unit is enabled.

The Second Vector Logical unit can be enabled and disabled through software by clearing bit 0 of word 3 in the Exchange Package of a user program. If the bit is clear, the unit is disabled and only the Full Vector Logical unit is available to instructions 140 through 145.

Because instruction 175 uses the Full Vector Logical unit, it cannot be chained with instructions 146 and 147, nor may it be chained with instructions 140 through 145 unless the Second Vector Logical unit is enabled and the instructions issue through that unit.

The Full Vector Logical functional unit time is 2 CPs; the Second Vector Logical functional unit time is 4 CPs.

Vector Population/Parity functional unit

The Vector Population/Parity functional unit counts the 1 bits in each element of the source V register. The total number of 1 bits is the population count. This population count can be an odd or an even number, as shown by its low-order bit.

Instructions 174*ij*1 (vector population count) and 174*ij*2 (vector population count parity) use the same operation code as the vector reciprocal approximation instruction. Some restrictions for the Reciprocal Approximation functional unit also apply for vector population instructions (see subsection on Reciprocal Approximation). The vector population count instruction delivers the total population count to elements of the destination V register.

The vector population count parity instruction delivers the low-order bit of the count to the destination V register. The Vector Population/Parity functional unit time is 5 CPs.

FLOATING-POINT FUNCTIONAL UNITS

Three floating-point functional units perform floating-point arithmetic for scalar and vector operations. When executing a scalar instruction, operands are obtained from S registers and results are delivered to an S register. When executing most vector instructions, operands are obtained from pairs of V registers or from an S register and a V register. Results are delivered to a V register. An exception is the Reciprocal Approximation unit requiring only one input operand.

Information on floating-point out-of-range conditions is contained in the subsection on Floating-point Arithmetic.

Floating-point Add functional unit

The Floating-point Add functional unit performs addition or subtraction of 64-bit operands in floating-point format and executes instructions 062, 063, and 170 through 173. A result is normalized even when operands are unnormalized. (Normalized floating-point numbers are described in the subsection on Floating-point Arithmetic.) Out-of-range exponents are detected as described in the subsection on Floating-point Arithmetic.

Floating-point Add functional unit time is 6 CPs.

Floating-point Multiply functional unit

The Floating-point Multiply functional unit executes instructions 064 through 067 and 160 through 167. These instructions provide for full- and half-precision multiplication of 64-bit operands in floating-point format and for computing two minus a floating-point product for reciprocal iterations.

The half-precision product is rounded; the full-precision product can be rounded or not rounded.

Input operands are assumed to be normalized. The Floating-point Multiply functional unit delivers a normalized result only if both input operands are normalized.

Out-of-range exponents are detected as described in the subsection on floating-point arithmetic. However, if both operands have zero exponents, the result is considered as an integer product, is not normalized, and is not considered out-of-range. This case provides a fast method of computing a 48-bit integer product, although the operands in this case must be shifted before the multiply operation.

Because the Second Vector Logical functional unit and the Floating-point Multiply functional units share input and output data paths, they cannot be used simultaneously. A reservation on one is a reservation on the other.

The Floating-point Multiply functional unit time is 7 CPs.

Reciprocal Approximation functional unit

The Reciprocal Approximation functional unit finds the approximate reciprocal of a 64-bit operand in floating-point format. The unit executes instructions 070 and 174*ij*0. Since the Vector Population/Parity functional unit shares some logic with this unit, the *k* designator must be 0 for the reciprocal approximation instruction to be recognized.

The input operand is assumed to be normalized and if so the result is correct. The high-order bit of the coefficient is not tested but is assumed to be a 1. Out-of-range exponents are detected as described under Floating-point Arithmetic.

The Reciprocal Approximation functional unit time is 14 CPs.

ARITHMETIC OPERATIONS

Functional units in a CPU perform either twos complement integer arithmetic or floating-point arithmetic.

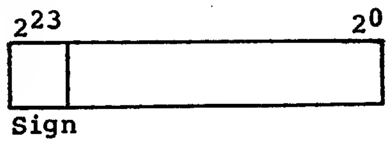
INTEGER ARITHMETIC

All integer arithmetic, whether 24 bits or 64 bits, is twos complement and is represented in the registers as illustrated in figure 4-4. The Address Add and Address Multiply functional units perform 24-bit arithmetic. The Scalar Add and the Vector Add functional units perform 64-bit arithmetic.

Multiplication of two scalar (64-bit) integer operands is accomplished by using the floating-point multiply instruction and one of the two methods that follows. The method used depends on the magnitude of the operands and the number of bits to contain the product.

If the operands are nonzero only in the 24 least significant bits, the two integer operands can be multiplied by shifting them each left 24 bits before the multiply operation. (The Floating-point Multiply functional unit recognizes the conditions where both operands have zero exponents as a special case.) The Floating-point Multiply functional unit returns the high-order 48 bits of the product of the coefficients as the coefficient of the result and leaves the exponent field 0. See figure 4-7. If the operand coefficients are generated by other than shifting so the low-order 24 bits would be nonzero, the low-order 48 bits of the product could have been nonzero, and the high-order 48 bits (the return part) could be one larger than expected as a truncation compensation constant is always added during a multiply.

Twos complement integer (24 bits)



Twos complement integer (64 bits)

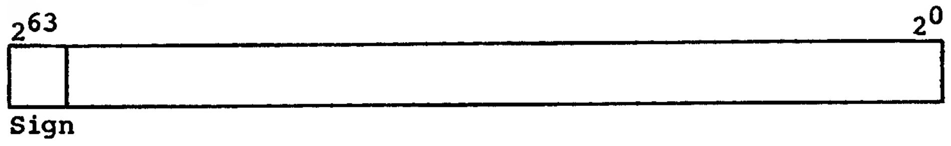


Figure 4-4. Integer data formats

If the operands are greater than 24 bits, multiplication is done by forming multiple partial products and then shifting and adding the partial products.

Division is done by algorithm; the particular algorithm used depends on the number of bits in the quotient. The quickest and most frequently used method is to convert the numbers to floating-point format and then use the floating-point functional units.

FLOATING-POINT ARITHMETIC

Floating-point numbers are represented in a standard format throughout the CPU. This format is a packed representation of a binary coefficient and an exponent (power of two). The coefficient is a 48-bit signed fraction. The sign of the coefficient is separated from the rest of the coefficient as shown in figure 4-5. Since the coefficient is signed magnitude, it is not complemented for negative values.

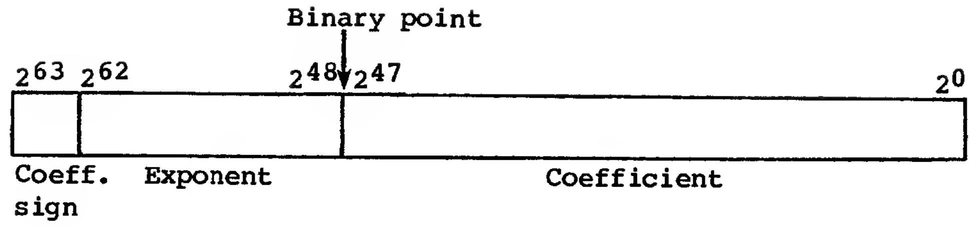


Figure 4-5. Floating-point data format

The exponent portion of the floating-point format is represented as a biased integer in bits 2^{62} through 2^{48}. The bias that is added to the exponents is 40000_8. The positive range of exponents is 40000_8 through 57777_8. The negative range of exponents is 37777_8 through 20000_8. Thus, the unbiased range of exponents is the following (note the negative range is one larger):

2^{-20000_8} through 2^{+17777_8}

In terms of decimal values, the floating-point format of the CRAY X-MP 4-processor allows the accurate expression of numbers to about 15 decimal digits in the approximate decimal range of 10^{-2466} through 10^{+2466}.

A zero value or an underflow result is not biased and is represented as a word of all zeros.

A negative 0 is not generated by any floating-point functional unit, except in the case where a negative 0 is one operand going into the Floating-point Multiply functional unit.

Normalized floating-point numbers, floating-point range errors, double-precision numbers, and the addition, multiplication, and division algorithms are described in the remainder of this subsection.

Normalized floating-point numbers

A nonzero floating-point number is normalized if the most significant bit of the coefficient is nonzero. This condition implies the coefficient has been shifted as far left as possible and the exponent adjusted accordingly. Therefore, the floating-point number has no leading zeros in the coefficient. The exception is that a normalized floating-point zero is all zeros.

When a floating-point number is created by inserting an exponent of 40060_8 into a 48-bit integer word, the result should be normalized before being used in a floating-point operation. Normalization is accomplished by adding the unnormalized floating-point operand to 0. Since S0 provides a 64-bit 0 when used in the S*j* field of an instruction, an operand in S*k* is normalized using the 062*i*0*k* instruction. S*i*, which can be S*k*, contains the normalized result.

The 170*i*0*k* instruction normalizes V*k* into V*i*.

Floating-point range errors

Overflow of the floating-point range is indicated by an exponent value of 60000_8 or greater in packed format. Detection of the overflow condition initiates an interrupt if the Floating-point Mode flag is set in the Mode register and monitor mode is not in effect. The Floating-point Mode flag can be set or cleared by a user mode program.

The Cray Operating System (COS) keeps a bit in a table to indicate the condition of the mode bit. System software manipulates the mode bit and uses the table bit to indicate how the mode should be left for the user. Therefore, the user usually needs to put the appropriate bit in the table if the user changes the mode.

Floating-point range error conditions are detected by the floating-point functional units as described in the following paragraphs.

Floating-point Add functional unit - A floating-point add range error condition is generated for scalar operands when the larger incoming exponent is greater than or equal to 60000_8. This condition sets the Floating-point Error flag with an exponent of 60000_8 being sent to the result register along with the computed coefficient, as in the following example:

```
 60000.4xxxxxxxxxxxxxxx   Range error
+57777.4xxxxxxxxxxxxxxx
 60000.6xxxxxxxxxxxxxxx   Result register
```

NOTE

If the result of an add or subtract operation is less than the machine minimum, the error is suppressed (even though both operands have exponents greater than or equal to 60000_8) because the machine minimum takes precedence in error detection.

Floating-point Multiply functional unit - Whether or not out-of-range conditions occur, and how they are handled, can be determined using the exponent matrix shown in figure 4-6. The exponent of the result, for any set of exponents, falls into one of seven unique zones. A description of each zone is given below.

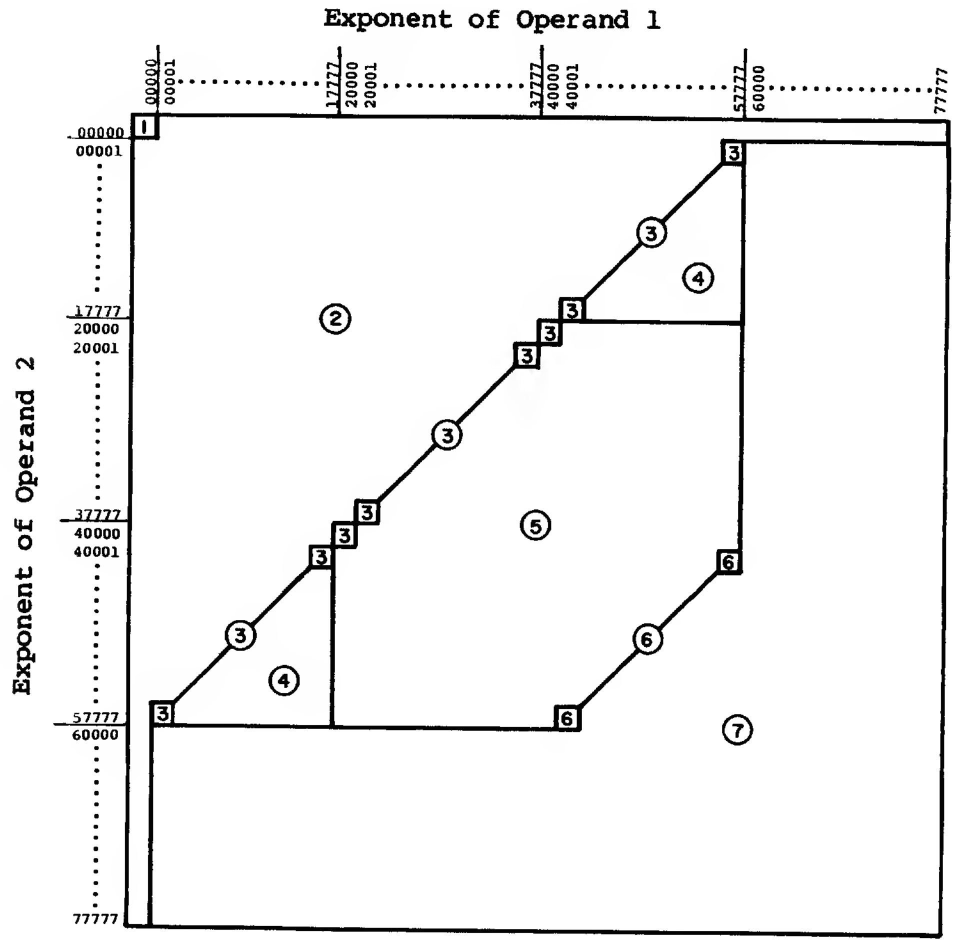


Figure 4-6. Exponent matrix for Floating-point Multiply unit

Zone	Description
1	Indicates a simple integer multiply; no fault is possible.
2	These exponents would result in an underflow condition. It is flagged as such, and the result is set to +0. (Multiply by 0 is in this group.)

Zone	Description
3	Underflow may occur on this boundary. The final exponent can be 17777_8 or 20000_8 depending on whether a normalized shift is required. If the exponent is 17777_8 and no normalized shift is required, the underflow will not be detected, and the coefficient and exponent will not be zeroed out. Underflow detection is done on the exponent used with the unshifted product coefficient.
4	The use of an underflow exponent is allowed if the final result is within the range 20000_8 to 57777_8.
5	This is the normal operand range and normal results are produced.
6	Overflow is flagged on this boundary. If a normalized shift is required, the value should be within bounds with a 57777_8 exponent. However, since overflow is detected using the exponent for the unnormalized shift condition (which is 60000_8), a 60000_8 will be inserted in the product as the final exponent.
7	Within this zone, an overflow fault is flagged and the product exponent is set to 60000_8.

NOTE

If either operand is less than the machine minimum, the error is suppressed (even though the other operand can be out of range) because the operand that is less than the machine minimum takes precedence in error detection.

Out-of-range conditions are tested before normalizing in the Floating-point Multiply functional unit. As shown above, if both incoming exponents are equal to 0, the operation is treated as an integer multiply. The result is treated normally with no normalization shift of the result allowed. The result is a 48-bit quantity starting with bit 2^{47}. When using this feature, the operands should be considered as 24-bit integers in bits 2^{47} through 2^{24}. In figure 4-6, if operand 1 is 4 and operand 2 is 6, a 48-bit result of 30_8 is produced. Bit 2^{63} obeys the usual rules for multiplying signs and the result is a sign and magnitude integer. Note the form of integers (see figure 4-4) accepted by the integer add and subtract and expected by the software is twos complement not sign and magnitude. Therefore, negative products must be converted.

If bits 2^0 through 2^{23} in operands 1 and 2 of figure 4-7 have any 1 bits, the product might be one (2^0) too large because a truncation compensation constant is added during the multiply process. (The following paragraphs discuss the truncation constant and its use.) The size of the shaded area in operands 1 and 2 (figure 4-7) does not need to be the same for both operands. To get a correct product, the only requirement is that the sum of the number of bits in the shaded area is 48 bits or more. If the sum is more than 48 bits, the binary point in the product is the number of places to the left that the sum is in excess of 48 (that is, assuming the operand binary points are at the left boundary of the shaded areas).

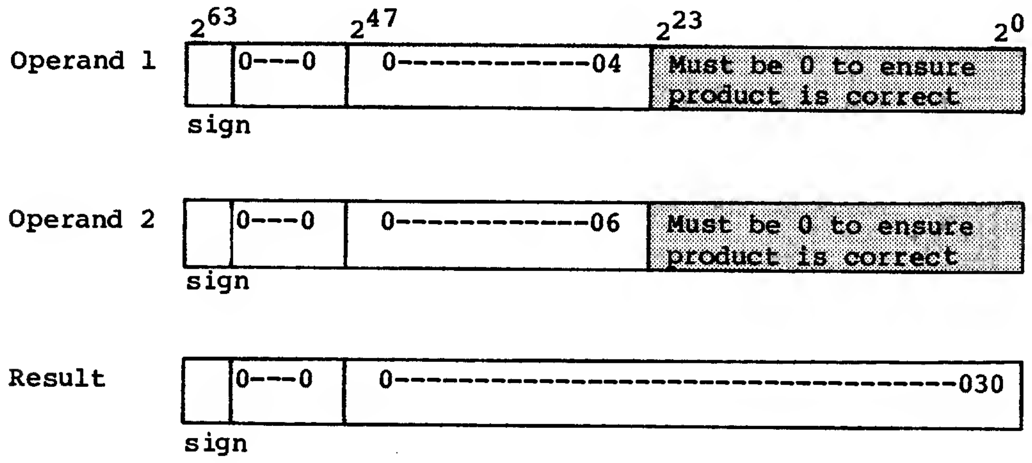


Figure 4-7. Integer multiply in Floating-point Multiply functional unit

Floating-point Reciprocal Approximation functional unit - For the Floating-point Reciprocal Approximation functional unit, an incoming operand with an exponent less than or equal to 20001_8 or greater than or equal to 60000_8 causes a floating-point range error. The error flag is set and an exponent of 60000_8 and the computed coefficient are sent to the result register.

Double-precision numbers

The CPU does not provide special hardware for performing double- or multiple-precision operations. Double-precision computations with 95-bit accuracy are available through software routines provided by Cray Research, Inc.

Addition algorithm

Floating-point addition or subtraction is performed in a 49-bit register (figure 4-8). Trial subtraction of the exponents selects the operand to be shifted down for aligning the operands. The larger exponent operand carries the sign. The coefficient of the number with the smaller exponent is shifted right to align with the coefficient of the number with the larger exponent. Bits shifted out of the register are lost; no roundup takes place. If the sum carries into the high-order bit, the low-order bit is discarded and an appropriate exponent adjustment is made. All results are normalized and if the result is less than the machine minimum, the error is suppressed.

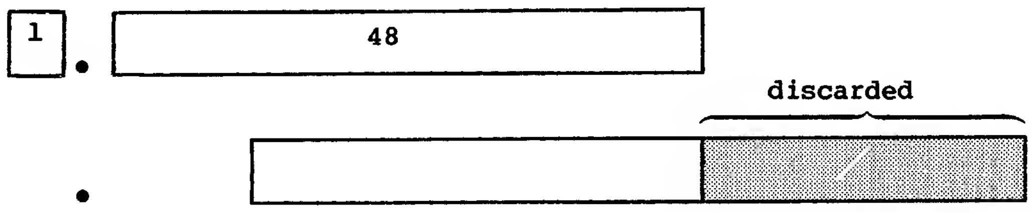


Figure 4-8. 49-bit floating-point addition

The Floating-point Add functional unit normalizes any floating-point number within the format of the Cray floating-point number system. The functional unit right shifts 1 or left shifts up to 48 per result to normalize the result.

One zero operand and one valid operand can be sent to the Floating-point Add functional unit, and the valid operand is sent through the unit normalized. Concurrently, the functional unit checks for overflow and/or underflow; underflow results are not flagged as errors.

Multiplication algorithm

The Floating-point Multiply functional unit has the two 48-bit coefficients as input into a multiply pyramid (see figure 4-9). If the coefficients are both normalized, then a full product is either 95 bits or 96 bits, depending on the value of the coefficients. A 96-bit product is normalized as generated. A 95-bit product requires a left shift of one to generate the final coefficient. If the shift is done, the final exponent is reduced by one to reflect the shift. The following discussion and the power of two designators used assumes that the product generated is in its final form; that is, no shift was required. On the system, the pyramid truncates part of the low-order bits of the 96-bit product. To adjust for this truncation, a constant is unconditionally added above the truncation. The average value of this truncation is 9.25×2^{-56}, which was determined by adding all carries produced by all possible combinations that could be truncated and dividing the sum by the number of possible combinations. Nine carries are injected at the 2^{-56} position to compensate for the truncated bits. The effect of the truncation without compensation is at most a result coefficient one smaller than expected. With compensation, the results range from one too large to one too small in the 2^{-48} bit position with approximately 99 percent of the values having zero deviation from what would have been generated had a full 96-bit pyramid been present. The multiplication is commutative; that is, A times B equals B times A.

Rounding is optional where truncation compensation is not. The rounding method used adds a constant so that it is 50 percent high ($.25 \times 2^{-48}$; high) 38 percent of the time and 25 percent low ($.125 \times 2^{-48}$; low) 62 percent of the time resulting in near zero average rounding error. In a full-precision rounded multiply, 2 round bits are entered into the pyramid at bit position 2^{-50} and 2^{-51} and allowed to propagate up the pyramid.

For a half-precision multiply, round bits are entered into the pyramid at bit positions 2^{-32} and 2^{-31}. A carry resulting from this entry is allowed to propagate up and the 29 most significant bits of the normalized result are transmitted back.

The variation due to this truncation and rounding are in the range:

$$-0.23 \times 2^{-48} \text{ to } +0.57 \times 2^{-48}$$

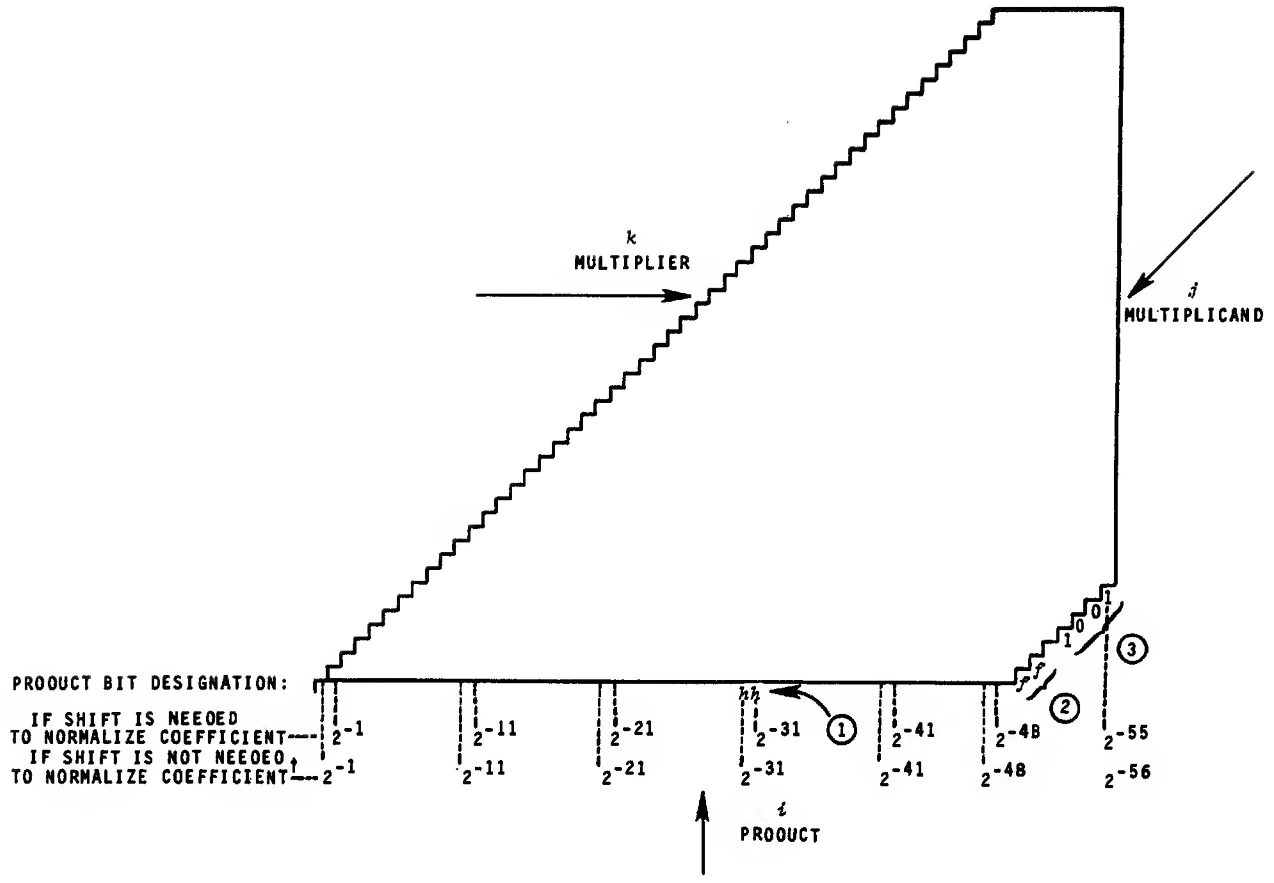
or $$-8.17 \times 10^{-16} \text{ to } +20.25 \times 10^{-16}.$$

With a full 96-bit pyramid and rounding equal to one-half the least significant bit, the variation would be expected to be:

$$-0.5 \times 2^{-48} \text{ to } +0.5 \times 2^{-48}$$

Division algorithm

The system performs floating-point division through reciprocal approximation, facilitating hardware implementation of a fully segmented functional unit. Because of this segmentation, operands enter the reciprocal unit during each CP. In vector mode, results are produced at a 1-CP rate and are used in other vector operations during chaining because all functional units in the system have the same result rate. The reciprocal approximation is based on Newton's method.



(1) *hh* = 11_2 for half-precision round, 00_2 for full-precision rounded or full-precision unrounded multiply

(2) *ff* = 11_2 for full-precision round, 00_2 for half-precision rounded or full-precision unrounded multiply

(3) Truncation compensation constant, 1001_2 used for all multiplies

Figure 4-9. Floating-point multiply partial-product sums pyramid

† Bit designations are used in the explanation of the Floating-point Multiply functional unit operation.

Newton's method - The division algorithm is an application of Newton's method for approximating the real roots of an arbitrary equation $F(x) = 0$, for which $F(x)$ must be twice differentiable with a continuous second derivative. The method requires making an initial approximation (guess), x_0, sufficiently close to the true root, x_t, being sought (see figure 4-10). For a better approximation, a tangent line is drawn to the graph of $y = F(x)$ at the point $(x_0, F(x_0))$. The X intercept of this tangent line is the better approximation x_1. This can be repeated using x_1 to find x_2, etc.

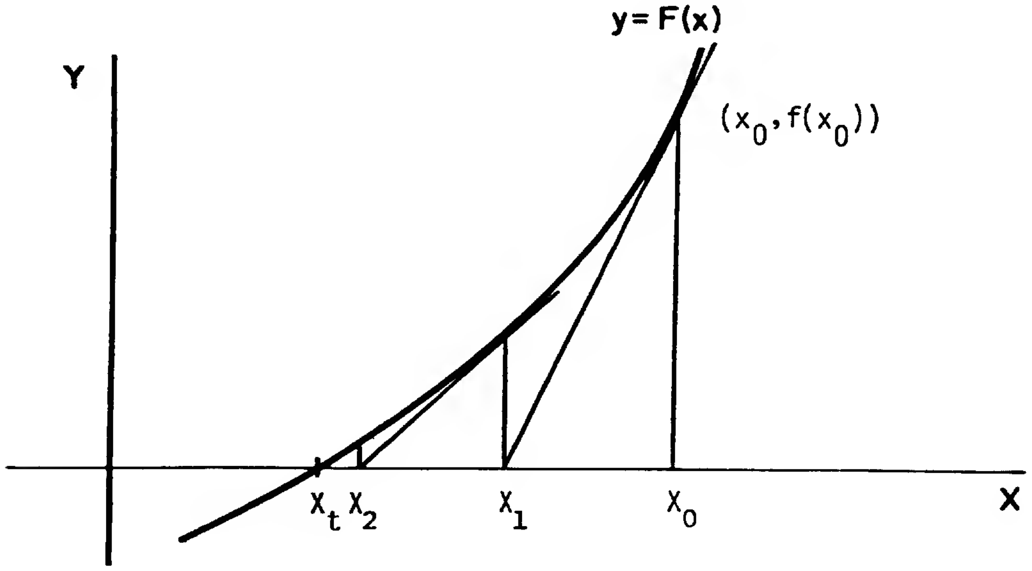


Figure 4-10. Newton's method

Derivation of the division algorithm

A definition for the derivative $F'(x)$ of a function $F(x)$ at point x_t is

$$F'(x_t) = \lim_{x \to x_t} \frac{F(x) - F(x_t)}{x - x_t}$$

if this limit exists. If the limit does not exist, $F(x)$ is not differentiable at the point t.

For any point x_i near to x_t,

$$F'(x_t) \approx \frac{F(x_i) - F(x_t)}{x_i - x_t}$$ where $\approx$ means "approximately equal to".

This approximation improves as x_i approaches x_t. Let x_i stand for an approximate solution and let x_t stand for the true answer being sought. The exact answer is then the value of x that makes F(x) equal 0. This is the case when $x=x_t$, therefore $F(x_t)$ in the equation above can be replaced by 0, giving the following approximation:

$$F'(x_t) \approx \frac{F(x_i)}{x_i - x_t} \qquad \text{Approximation (1)}$$

Notice that $x_t - x_i$ is the correction applied to an approximate answer, x_i, to give the right answer since $x_i + (x_t - x_i)$ equals x_t. Solving approximation (1) for $(x_t - x_i)$ gives:

$$x_t - x_i = \text{correction} \approx - \frac{F(x_i)}{F'(x_t)},$$

that is, $-\dfrac{F(x_i)}{F'(x_t)}$ is the approximate correction.

If this quantity is substituted into the approximation, then:

$$x_t \approx (x_i + \text{approximate correction}) = x_{i+1}.$$

This gives, the following equation:

$$x_{i+1} = x_i - \frac{F(x_i)}{F'(x_i)}, \qquad \text{Equation (1)}$$

where x_{i+1} is a better approximation than x_i to the true value, x_t, being sought. The exact answer is generally not obtained at once because the correction term is not generally exact. However, the operation is repeated until the answer becomes sufficiently close for practical use.

To make use of Newton's method to find the reciprocal of a number B, simply use F(x) = (1/x - B).

First calculating F'(x):

where $F'(x) = (\frac{1}{x} - B)' = (\frac{-1}{x^2})$. thus for any point $x_1 \neq 0$,

$F'(x_1) = -\dfrac{1}{x_1^2}$. Choosing for x, a value near $\dfrac{1}{B}$

and applying equation (1),

$$x_2 = x_1 - \frac{\frac{1}{x_1} - B}{-\frac{1}{x_1^2}},$$

$$x_2 = x_1 + x_1^2 \left(\frac{1}{x_1} - B\right),$$

$$x_2 = x_1 + x_1 - x_1^2 B,$$

$$x_2 = 2x_1 - x_1^2 B = x_1(2-x_1 B).$$

On the system, x_1 times the quantity in parentheses is performed by a floating-point multiply. $2-x_1B$ is performed by the reciprocal approximation instruction. x_1 is the x near 1/B and is formed by the half-precision reciprocal approximation instruction.

This approximation technique using Newton's method is implemented in the system. A hardware table look up provides an initial guess, x_0, to start the process.

$x_0(2 - x_0B)$	1st approximation, I1	Done in reciprocal unit
$x_1(2 - x_1B)$	2nd approximation, I2	
$x_2(2 - x_2B)$	3rd approximation, I3	
$x_3(2 - x_3B)$	4th approximation	Done with software

The system's Reciprocal Approximation functional unit performs three iterations: I1, I2, and I3. I1 is accurate to 8 bits and is found after a table look-up to choose the initial guess, x_0. I2 is the second iteration and is accurate to 16 bits. I3 is the final (third) iteration answer of the Reciprocal Approximation functional unit, and its result is accurate to 30 bits.

A fourth iteration uses a special instruction within the Floating-point Multiply functional unit to calculate the correction term. This iteration is used to increase accuracy of the reciprocal unit's answer to full precision. A fifth iteration should not be done.

The division algorithm that computes S1/S2 to full-precision requires the following operations:

S3 = 1/S2	Performed by the Reciprocal Approximation functional unit
S4 = (2 - (S3 * S2))	Performed by the Floating-point Multiply functional unit in iteration mode
S5 = S4 * S3	Performed by the Floating-point Multiply functional unit using full-precision. S5 now equals 1/S2 to 48-bit accuracy.
S6 = S5 * S1	Performed by the Floating-point Multiply functional unit using full-precision rounded

The reciprocal approximation at step 1 is correct to 30 bits. An additional Newton iteration (fourth iteration) at operations 2 and 3 increases this accuracy to 48 bits. This iteration answer is applied as an operand in a full-precision rounded multiply operation to obtain the quotient accurate to 48 bits. Additional iterations should not be attempted since erroneous results are possible.

****

CAUTION

The reciprocal iteration is designed for use once with each half-precision reciprocal generated. If the fourth iteration (the programmed iteration) results in an exact reciprocal or if an exact reciprocal is generated by some other method, performing another iteration results in an incorrect final reciprocal.

****

Where 29 bits of accuracy are sufficient, the reciprocal approximation instruction is used with the half-precision multiply to produce a half-precision quotient in only two operations.

S3 = 1/S2	Performed by the Reciprocal Approximation functional unit
S6 = S1 * S3	Performed by the Floating-point Multiply functional unit in half-precision

The 19 low-order bits of the half-precision results are returned as zeros with a rounding applied to the low-order bit of the 29-bit result.

Another method of computing divisions is as follows:

S3 = 1/S2	Performed by the Reciprocal Approximation functional unit
S5 = S1 * S3	Performed by the Floating-point Multiply functional unit
S4 = (2 - (S3 * S2))	Performed by the Floating-point Multiply functional unit
S6 = S4 * S5	Performed by the Floating-point Multiply functional unit

A scalar quotient is computed in 29 CPs since operations 2 and 3 issue in successive CPs. With this method, the correction to reach a full-precision reciprocal is applied after the numerator is multiplied times the half-precision reciprocal rather than before.

A vector quotient using this procedure requires less than four vector times since operations 1 and 2 are chained together. This overlaps one of the multiply operations. (A vector time is 1 CP for each element in the vector.)

****

CAUTION

The coefficient of the reciprocal produced by the alternate method can be as much as 2×2^{-48} different from the first method described for generating full-precision reciprocals. This difference can occur because one method can round up as much as twice while the other method may not round at all. One round can occur while the correction is generated and the second round can occur when producing the final quotient.

Therefore, if the reciprocals are to be compared, the same method should be used each time the reciprocals are generated. Cray FORTRAN (CFT) uses a consistent method and ensures the reciprocals of numbers are always the same.

****

For example, two 64-element vectors are divided in 3 * 64 CPs plus overhead. (The overhead associated with the functional units for this case is 38 CPs.)

LOGICAL OPERATIONS

Scalar and vector logical units perform bit-by-bit manipulation of 64-bit quantities. Operations provide for forming logical products, differences, sums, and merges.

A logical product is the AND function:

Operand 1	1 0 1 0
Operand 2	1 1 0 0
Result	1 0 0 0

An operation similar to the AND function produces the following results:

Operand 1	1 0 1 0
Operand 2	1 1 0 0
Result	0 1 0 0

The logical product (AND) operation is used for masking operations where the ones specify the bits to be saved. In this variant of the AND function, the zeros specify the bits to be saved (Operand 1 is the mask).

A logical sum is the inclusive OR function:

Operand 1	1 0 1 0
Operand 2	1 1 0 0
Result	1 1 1 0

A logical difference is the exclusive OR function:

Operand 1	1 0 1 0
Operand 2	1 1 0 0
Result	0 1 1 0

A logical equivalence is the exclusive NOR function:

Operand 1	1 0 1 0
Operand 2	1 1 0 0
Result	1 0 0 1

The merge uses two operands and a mask to produce results as follows:

Operand 1	1	0	1	0	1	0	1	0
Operand 2	1	1	0	0	1	1	0	0
Mask	1	1	1	1	0	0	0	0
Result	1	0	1	0	1	1	0	0

The bits of operand 1 pass where the mask bit is 1. The bits of operand 2 pass where the mask bit is 0.

CPU INSTRUCTIONS 5

INSTRUCTION FORMAT

Each instruction used in the computer is either a 1-parcel (16-bit) instruction or a 2-parcel (32-bit) instruction. Instructions are packed four parcels per word. Parcels in a word are numbered 0 through 3 from left to right and any parcel position can be addressed in branch instructions. A 2-parcel instruction begins in any parcel of a word and can span a word boundary. For example, a 2-parcel instruction beginning in the fourth parcel of a word ends in the first parcel of the next word. No padding to word boundaries is required. Figure 5-1 illustrates the general form of instructions.

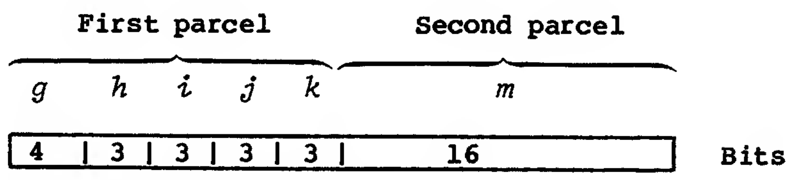


Figure 5-1. General form for instructions

Four variations of this general format use the fields differently; two forms are 1-parcel formats and two are 2-parcel formats. The formats of these four variations are described below.

1-PARCEL INSTRUCTION FORMAT WITH DISCRETE *j* AND *k* FIELDS

The most common of the 1-parcel instruction formats uses the *i*, *j*, and *k* fields as individual designators for operand and result registers (see figure 5-2). The *g* and *h* fields define the operation code. The *i* field designates a result register and the *j* and *k* fields designate operand registers. Some instructions ignore one or more of the *i*, *j*, and *k* fields. The following types of instructions use this format.

- Arithmetic
- Logical
- Double shift
- Floating-point constant

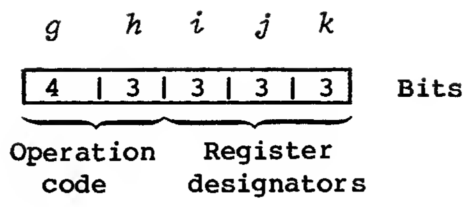


Figure 5-2. 1-parcel instruction format with discrete *j* and *k* fields

1-PARCEL INSTRUCTION FORMAT WITH COMBINED *j* AND *k* FIELDS

Some 1-parcel instructions use the *j* and *k* fields as a combined 6-bit field (see figure 5-3). The *g* and *h* fields contain the operation code, and the *i* field is generally a destination register identifier. The combined *j* and *k* fields generally contain a constant or a B or T register designator. The branch instruction 005 and the following types of instructions use the 1-parcel instruction format with combined *j* and *k* fields.

- Constant
- B and T register block memory transfer
- B and T register data transfer
- Single shift
- Mask

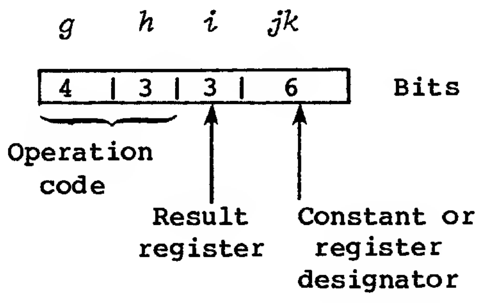


Figure 5-3. 1-parcel instruction format with combined *j* and *k* fields

2-PARCEL INSTRUCTION FORMAT WITH COMBINED *j*, *k*, AND *m* FIELDS

The instruction type for a 22-bit immediate constant uses the combined *j*, *k*, and *m* fields to hold the constant. The 7-bit *gh* field contains an operation code, and the 3-bit *i* field designates a result register. The instruction type using this format transfers the 22-bit *jkm* constant to an A or S register.

The instruction type used for scalar memory transfers also requires a 22-bit *jkm* field for an address displacement. This instruction type uses the 4-bit *g* field for an operation code, the 3-bit *h* field to designate an address index register, and the 3-bit *i* field to designate a source or result register. (See subsection on Special Register Values.)

Figure 5-4 shows the two general applications for the 2-parcel instruction format with combined *j*, *k*, and *m* fields.

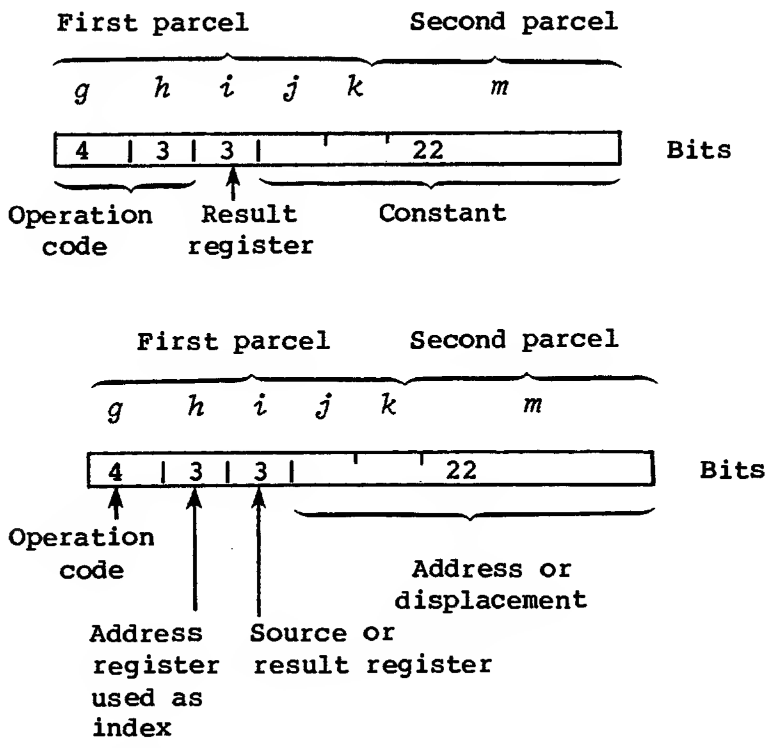


Figure 5-4. 2-parcel instruction format with combined *j*, *k*, and *m* fields

2-PARCEL INSTRUCTION FORMAT WITH COMBINED *i*, *j*, *k*, AND *m* FIELDS

The 2-parcel instruction type for a branch (figure 5-5) uses the combined *i*, *j*, *k*, and *m* fields to contain the 24-bit address that allows branching to an instruction parcel. A 7-bit operation code (*gh*) is followed by an *ijkm* field. The high-order bit of the *i* field is clear.

The 2-parcel instruction type for a 24-bit immediate constant (figure 5-6) uses the combined *i*, *j*, *k*, and *m* fields to hold the constant. This instruction type uses the 4-bit *g* field for an operation code and the 3-bit *h* field to designate the result address register. The high-order bit of the *i* field is set.

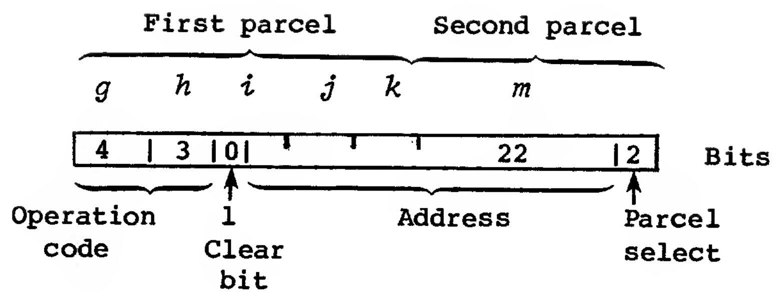


Figure 5-5. 2-parcel instruction formats for a branch with combined *i*, *j*, *k*, and *m* fields

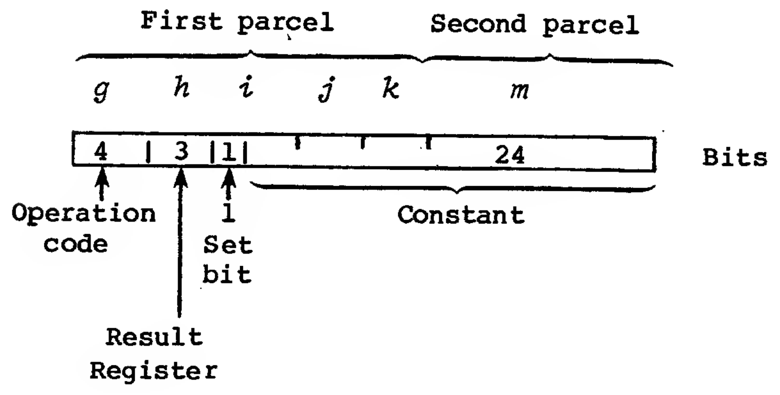


Figure 5-6. 2-parcel instruction formats for a 24-bit immediate constant with combined *i*, *j*, *k*, and *m* fields

SPECIAL REGISTER VALUES

If the S0 and A0 registers are referenced in the *j* or *k* fields of an instruction, the contents of the respective register are not used; instead, a special operand is generated. The special value is available regardless of existing A0 or S0 reservations (and in this case are not checked). This use does not alter the actual value of the S0 or A0 register. If S0 or A0 is used in the *i* field as the operand, the actual value of the register is provided. The table below shows the special register values.

Field	Operand value
A*h*, *h*=0	0
A*i*, *i*=0	(A0)
A*j*, *j*=0	0
A*k*, *k*=0	1
S*i*, *i*=0	(S0)
S*j*, *j*=0	0
S*k*, *k*=0	2^{63}

INSTRUCTION ISSUE

Instructions are read one parcel at a time from the instruction buffers and delivered to the Next Instruction Parcel (NIP) register. The instruction is then passed to the Current Instruction Parcel (CIP) register when the previous instruction issues. An instruction in the CIP register issues when conditions in the functional unit and registers are such that functions required for execution can be performed without conflicting with a previously issued instruction. Instruction parcels can issue out of the CIP register at a maximum rate of one per clock period.

Execution times (the time from issue to delivery of data to the destination operating registers) are fixed for instructions 000 through 077, except those that reference memory (instructions 000, 004, branch instructions 005 through 017, and block transfer instructions 034 through 037). Scalar memory instructions 100 through 137 complete in variable lengths of time. Vector operation instructions 140 through 177 complete in a fixed time if the instructions are not chained to memory fetches.

Execution times can be affected by instruction 0034*jk*, which tests and sets the semaphore designated by *jk*. If the semaphore is set, instruction issue is held until another CPU clears that semaphore. If the semaphore is clear, the instruction issues and sets the semaphore. If all CPUs in a cluster are holding issue on a test and set, a flag is set in the Exchange Package (if not in monitor mode) and an exchange occurs. If an interrupt occurs while a test and set instruction is holding in the CIP register, a flag is set in the Exchange Package, CIP and NIP registers clear, and an exchange occurs with the P register pointing to the test and set instruction.

Entry to the NIP register is blocked for the second parcel of a 2-parcel instruction, leaving NIP blanked. Instead, the parcel is delivered to the Lower Instruction Parcel (LIP) register. The zeros in NIP (the pseudo second parcel) are transferred to CIP and issued as a do-nothing instruction.

When special register values (A0 or S0) are selected by an instruction for A*h*, A*j*, A*k*, S*j*, or S*k*, the normal "hold issue until operand ready" conditions do not apply. These values are always immediately available.

INSTRUCTION DESCRIPTIONS

This section contains detailed information about individual instructions or groups of related instructions. Each instruction begins with boxed information consisting of the Cray Assembly Language (CAL) syntax format, a brief description of each instruction, and the octal code sequence defined by the *gh* fields. The appearance of an *m* in a format designates an instruction consisting of two parcels.

Following the boxed information is a more detailed description of the instruction or instructions, including a list of hold issue conditions, execution time, and special cases. Hold issue conditions refer to those conditions delaying issue of an instruction until conditions are met.

Instruction issue time assumes that if an instruction issues at clock period *n* (CP *n*), the next instruction issues at CP *n* + issue time† if its own issue conditions have been met.

The following special characters can appear in the operand field description of symbolic machine instructions and are used by the assembler in determining the operation to be performed.

+	Arithmetic sum of adjoining registers
-	Arithmetic difference of adjoining registers
*	Arithmetic product of adjoining registers
/	Division or reciprocal
#	Use ones complement
>	Shift value or form mask from left to right
<	Shift value or form mask from right to left
&	Logical product of adjoining registers
!	Logical sum of adjoining registers
\	Logical difference of adjoining registers

† Previous instruction issued

In some instructions, register designators are prefixed by the following letters, which have special meaning to the assembler.

F	Floating-point operation
H	Half-precision operation
R	Rounded operation
I	Reciprocal iteration
P	Population count
Q	Population count parity
Z	Leading zero count

INSTRUCTION 000

CAL Syntax	Description	Octal Code
ERR	Error exit	000000

Instruction 000 is treated as an error condition and an exchange sequence occurs. Content of the instruction buffers is voided by the exchange sequence. Instruction 000 halts execution of an incorrectly coded program branching into an unused area of memory (if memory was backgrounded with zeros) or into a data area (if the data is positive integers, right-justified ASCII, or floating-point zero). If monitor mode is not in effect, the Error Exit flag in the F register is set. All instructions issued before this instruction are run to completion. When results of previously issued instructions arrive at the operating registers, an exchange occurs to the Exchange Package designated by contents of the XA register. The program address stored during the exchange on the terminating exchange sequence is the contents of the P register advanced by one count (that is, the address of the instruction following the error exit instruction).

HOLD ISSUE CONDITIONS:	Any A, S, or V register reserved
EXECUTION TIME:	Instruction issue, 40 CPs; this time includes an exchange sequence (24 CPs) and a fetch operation (16 CPs).
SPECIAL CASES:	None

CAL Syntax	Description	Octal Code
CA,A*j* A*k*	Set the Current Address (CA) register for the channel indicated by (A*j*) to (A*k*) and activate the channel	0010*jk*
CL,A*j* A*k*	Set the Limit Address (CL) register for the channel indicated by (A*j*) to (A*k*)	0011*jk*
CI,A*j*	Clear the interrupt flag and error flag for the channel indicated by (A*j*); clear device master-clear (output channel)	0012*j*0
MC,A*j*	Clear the interrupt flag and error flag for the channel indicated by (A*j*); set device master-clear (output channel); clear device ready-held (input channel)	0012*j*1
XA A*j*	Enter the XA register with (A*j*)	0013*j*0

Instructions 0010 through 0013 are privileged to monitor mode and provide operations useful to the operating system. Functions are selected through the *i* designator. Instructions are treated as pass instructions if the monitor mode bit is not set.

When the *i* designator is 0, 1, or 2, the instruction controls operation of the I/O channels. Each channel has two registers directing the channel activity. The CA register for a channel contains the address of the current channel word. The CL register specifies the limit address. In programming the channel, the CL register is initialized first and then CA sets, activating the channel. As transfer continues, CA is incremented toward CL. When (CA) is equal to (CL), transfer is complete for words at initial (CA) through (CL)-1. When the *j* designator is 0 or when the 5 low-order bits of A*j* are less than 6_8, the functions are executed as pass instructions. Valid channel numbers are $6\text{-}17_8$. When the *k* designator is 0, CA or CL is set to 1.

When the *i* designator is 3, the instruction transmits bits 2^{11} through 2^4 of (A*j*) to the XA register. When the *j* designator is 0, the XA register is cleared.

Instruction 0012*j*0 is used to clear the device Master Clear. For instruction 0012, if the *k* designator is 1 for an output channel, the master clear is set; if the *k* designator is 1 for an input channel, the Ready flag is cleared.

HOLD ISSUE CONDITIONS:	For instructions 0010 and 0011, A*j* or A*k* reserved (except A0) For instructions 0012 or 0013, A*j* reserved (except A0)
EXECUTION TIME:	Instruction issue, 1 CP
SPECIAL CASES:	If the program is not in monitor mode, the instruction becomes a no-op although all hold issue conditions remain effective.

For instructions 0010, 0011, and 0012:
If *j*=0, the instruction is a no-op.
If *k*=0, CA or CL is set to 1.
If 5 low-order bits of (A*j*) are less than 6_8, the instruction is a no-op. If the 5 low-order bits of (A*j*) are greater than 17_8, undetermined results can occur. (That is, 6_8 through 17_8 are valid, 20_8 through 37_8 are undetermined, 46_8 through 57_8 are valid, etc.)

For instruction 0012:
The correct priority interrupting channel number cannot be read (through instruction 033) until 6 CPs after issue of instruction 0012.

For instruction 0013:
If *j*=0, XA register is cleared.

NOTE

Because there is no hardware interlock among CPUs, it is possible to have more than one CPU issuing these instructions at the same time; however, undetermined results will occur.

Software must ensure only one CPU is servicing I/O at a time while in monitor mode.

INSTRUCTION 0014

CAL Syntax	Description	Octal Code
RT S*j*	Enter the Real-time Clock register with (S*j*)	0014*j*0
IP,*j*1	Set interprocessor interrupt request of CPU*j*	0014*j*1
IP 0	Clear received interprocessor interrupt request from all other processors	001402
CLN 0	Cluster number = 0	001403
CLN 1	Cluster number = 1	001413
CLN 2	Cluster number = 2	001423
CLN 3	Cluster number = 3	001433
CLN 4	Cluster number = 4	001443
CLN 5	Cluster number = 5	001453
PCI S*j*	Enter Interrupt Interval (II) register with (S*j*)	0014*j*4
CCI	Clear the programmable clock interrupt request	001405
ECI	Enable programmable clock interrupt request	001406
DCI	Disable programmable clock interrupt request	001407

Instruction 0014 performs specialized functions for managing the real-time and programmable clocks and handles interprocessor interrupt requests and cluster number operations. Instruction 0014 is privileged to monitor mode and is treated as a pass instruction if the monitor mode bit is not set.

When the *k* designator is 0, the instruction loads the contents of the S*j* register into the RTC register. When the *j* designator is 0 or (S*j*)=0, the RTC register is cleared.

When the *k* designator is 1, the instruction sets the internal CPU interrupt request in the CPU associated with PN=*j*. If the CPU associated with PN=*j* is not in monitor mode, the Interrupt from Internal CPU (ICP) flag sets in the F register causing an interrupt. The request remains until cleared by the receiving CPU issuing instruction 001402. If the CPU associated with PN=*j* attempts to interrupt itself, the instruction becomes a no-op.

When the *k* designator is 2, the instruction clears the internal CPU interrupt request set by any other CPU.

When the *k* designator is 3, the instruction sets the cluster number to *j* to make the following cluster selections:

CLN = 0	No cluster; all shared register and semaphore operations are no-ops, (except SB, ST, or SM register reads, which return a 0 value to A*i* or S*i*).
CLN = 1	Cluster 1
CLN = 2	Cluster 2
CLN = 3	Cluster 3
CLN = 4	Cluster 4
CLN = 5	Cluster 5

Clusters 1, 2, 3, 4 and 5 each have a separate set of SM, SB, and ST registers.

When the *k* designator is 4, the instruction loads the low-order 32 bits from the S*j* register into both the II register and the ICD counter. When the *j* designator is 0 or (S*j*)=0, II and ICD are cleared.

When the *k* designator is 5, the instruction clears the programmable clock interrupt request if the request is previously set by ICD counting down to 0.

When the *k* designator is 6, the instruction enables repeated programmable clock interrupt requests at a repetition rate determined by the value stored in the II register.

When the *k* designator is 7, the instruction disables repeated programmable clock interrupt requests until an instruction 001406 is executed to enable the requests.

HOLD ISSUE CONDITIONS:	S*j* reserved (except S0)
	For instruction 0014*j*3, hold issue 2+† CPs
EXECUTION TIME:	Instruction issue, 1 CP
SPECIAL CASES:	If the program is not in monitor mode, these instructions become no-ops but all hold issue conditions remain effective.
	For instructions 0014*j*0 and 0014*j*4, if *j*=0, (S*j*)=0.
	For instruction 0014*j*0, the value is entered into the RTC register 4 CPs after instruction 0014*j*0 issues.
	For instruction 0014*j*1, if the processor number equals *j* of the CPU issuing this instruction, the instruction becomes a no-op. (A CPU cannot interrupt itself if *j* equals the processor number of the CPU issuing this instruction.)

† If more than one CPU attempts to access semaphores or shared registers in the same clock period, a scanner will resolve the conflict. See shared register explanation in section 2.

INSTRUCTION 0015

CAL Syntax	Description	Octal Code
†	Select performance monitor	0015*j*0
†	Set maintenance read mode	001501
†	Load diagnostic checkbyte with S1	001511
†	Set maintenance write mode 1	001521
†	Set maintenance write mode 2	001531

These instructions are all privileged to monitor mode.

Instruction 0015*j*0 selects one of four groups of hardware related events to be monitored by the performance counters. See Appendix C for a description of how performance monitoring is accomplished.

Instructions 001501 through 001531 are used to check the operation of the modules concerned with SECDED and to verify error detection and correction. The maintenance mode switch on the mainframe's control panel must be switched on during execution of these instructions or they become no-ops. See Appendix D for a description of SECDED maintenance mode functions.

Instructions 001501 and 001521 are used to verify check bit memory storage. Instruction 001501 allows the 8 check bits for SECDED to replace certain data bit positions in any subsequent memory read for the CPU path (including fetch and I/O). Instruction 001521 allows certain write data bits to replace the 8 check bits for SECDED for any subsequent CPU write to memory.

Instructions 001511 and 001531 are used to verify error detection and correction. Instruction 001511 loads a diagnostic check byte with the high order 8 bits of S1. Instruction 001531 enables a diagnostic check byte to replace the 8 check bits for SECDED being written into memory for any subsequent write to memory.

† Not supported at this time

INSTRUCTION 0020

CAL Syntax		Description	Octal Code
VL	A*k*	Transmit (A*k*) to VL register	00200*k*
VL	1[†]	Transmit 1 to VL register	002000

Instruction 00200*k* enters the VL register with a value determined by the contents of A*k*. The low-order 6 bits of (A*k*) are entered into the VL register. The 7th bit of VL is set if the 6 low-order bits of (A*k*)=0.

For example, if (A*k*)=0 or a multiple of 100_8, then VL=100_8. The content of VL is always between 1 and 100_8.

Instruction 002000 transmits the value of 1 to the VL register.

HOLD ISSUE CONDITIONS:	A*k* reserved (except A0)
EXECUTION TIME:	Instruction issue, 1 CP VL register ready, 1 CP
SPECIAL CASES:	Maximum vector length is 64. (A*k*)=1 if *k*=0. (VL)=100_8 if *k*≠0 and (A*k*)=0 or a multiple of 100_8.

† Special CAL syntax

CAL Syntax	Description	Octal Code
EFI	Enable interrupt on floating-point error	002100
DFI	Disable interrupt on floating-point error	002200
ERI	Enable interrupt on operand (address) range error	002300
DRI	Disable interrupt on operand (address) range error	002400
DBM	Disable bidirectional memory transfers	002500
EBM	Enable bidirectional memory transfers	002600
CMR	Complete memory references	002700

Instruction 002100 sets the Floating-point Mode flag in the M register. Instruction 002200 clears the Floating-point Mode flag in the M register. The two instructions do not check the previous state of the flag. When set, the Floating-point Mode flag enables interrupts on floating-point range errors as described in section 4. Issuing either of these instructions also clears the Floating-Point Error Status flag.

Instruction 002300 sets the Operand Range Mode flag in the M register. Instruction 002400 clears the Operand Range Mode flag in the M register. The two instructions do not check the previous state of the flag. When set, the Operand Range Mode flag enables interrupts on operand (address) range errors as described in section 3.

Instruction 002500 disables the bidirectional memory mode. Instruction 002600 enables the bidirectional memory mode. Block reads and writes can operate concurrently in bidirectional memory mode. If the bidirectional memory mode is disabled, only block reads can operate concurrently.

Instruction 002700 assures completion of all memory references within a particular CPU issuing the instruction. Instruction 002700 does not issue until all memory references before this instruction are at the stage of execution where completion occurs in a fixed amount of time. For example, a load of any data that has been stored by the CPU issuing instruction CMR, 002700 is assured of receiving the updated data if the load is issued after the CMR instruction. Synchronization of memory references between processors can be done by this instruction in conjunction with semaphore instructions.

HOLD ISSUE CONDITIONS:	Instructions 002500 and 002600, hold issue 2 CPs Instruction 002700, Ports A, B, C busy Instruction 002700, scalar memory reference active in clock period 1, 2, or 3 A*k* reserved (except A0)
EXECUTION TIME:	Instruction issue, 1 CP
SPECIAL CASES:	Instructions 002100 and 002200 are issued even if there are other floating-point operations in process resulting from previous issues. The interrupts are enabled or disabled at CP + 1; floating-point overflows occurring after that time cause interrupts if they are enabled even if the overflow is generated by a previously issued floating-point instruction. Instructions 002300 and 002400 are issued even if there are other memory references in process resulting from previous issues. The interrupts are enabled or disabled at CP + 1; operand range errors occurring after that time cause interrupts if they are enabled even if the operand range error is generated by a previous memory reference.

INSTRUCTIONS 0030, 0034, 0036, and 0037

CAL Syntax	Description	Octal Code
VM S*j*	Transmit (S*j*) to VM register	0030*j*0
VM 0[†]	Clear VM register	003000
SM*jk* 1,TS	Test and set semaphore *jk*, $0 \le jk \le 31_{10}$	0034*jk*
SM*jk* 0	Clear semaphore *jk*, $0 \le jk \le 31_{10}$	0036*jk*
SM*jk* 1	Set semaphore *jk*, $0 \le jk \le 31_{10}$	0037*jk*

Instruction 0030*j*0 enters the VM register with the contents of S*j*. The VM register is cleared if the *j* designator is 0 in instruction 003000. These instructions are used in conjunction with the vector merge instructions (146 and 147) in which an operation is performed depending on the contents of VM.

Instruction 0034*jk* tests and sets the semaphore designated by *jk*. If the semaphore is set, issue is held until the other CPU clears that semaphore. If the semaphore is clear, the instruction issues and sets the semaphore. If all CPUs in a cluster are holding issue on a test and set, the DL flag is set in the Exchange Package (if not in monitor mode) and an exchange occurs. If an interrupt occurs while a test and set instruction is holding in the CIP register, the WS flag in the Exchange Package sets, CIP and NIP registers clear, and an exchange occurs with the P register pointing to the test and set instruction. The SM register is 32 bits with SM0 being the most significant bit.

Instruction 0036*jk* clears the semaphore designated by *jk*.

Instruction 0037*jk* sets the semaphore designated by *jk*.

HOLD ISSUE CONDITIONS: For instruction 0030*j*0:
- S*j* reserved (except S0)
- Instruction 003 in process, unit busy 1 CP
- Instruction 14*x* in process, unit busy (VL) + 5 CPs
- Instruction 175 in process, unit busy (VL) + 5 CPs

† Special CAL syntax

HOLD ISSUE CONDITIONS: (continued)	For instructions 0034*jk*, 0036*jk*, and 0037*jk*: Hold issue 1+ CP[†]
	For instruction 0034*jk*: If current Cluster Number≠0 and SM*jk* is set, holds issue until other CPU in the same cluster clears the semaphore.
EXECUTION TIME:	Instruction issue, 1 CP
SPECIAL CASES:	(S*j*)=0 if *j*=0.
	Instructions 0034*jk*, 0036*jk*, and 0037*jk* are no-ops if CLN=0.

† If more than one CPU attempts to access semaphores or shared registers in the same clock period, a scanner will resolve the conflict. See shared register explanation in section 2.

INSTRUCTION 004

CAL Syntax	Description	Octal Code
EX	Normal exit	004000

Instruction 004 causes an exchange sequence which voids the contents of the instruction buffers. If monitor mode is not in effect, the Normal Exit flag in the F register is set. All instructions issued before this instruction are run to completion; that is, when all results arrive at the operating registers because of previously issued instructions, an exchange sequence occurs to the Exchange Package designated by the contents of the XA register. The program address stored into the Exchange Package is advanced one count from the address of the normal exit instruction. Instruction 004 is used to issue a monitor request from a user program.

HOLD ISSUE CONDITIONS:	Any A, S, or V register reserved
EXECUTION TIME:	Instruction issue, 40 CPs; this time includes an exchange sequence (24 CPs) and a fetch operation (16 CPs).
SPECIAL CASES:	None

INSTRUCTION 005

CAL Syntax	Description	Octal Code
J B*jk*	Branch to (B*jk*)	0050*jk*

Instruction 005 sets the P register to the 24-bit parcel address specified by the contents of B*jk* causing execution to continue at that address. The instruction is used to return from a subroutine.

HOLD ISSUE CONDITIONS: Instruction 034 or 035 in process

Instruction 025 issued in the previous CP

Second parcel in a different buffer, 2 CP delay

Second parcel not in a buffer

EXECUTION TIME: Instruction issue:

Instruction parcel and following parcel both in a buffer and branch address in a buffer, 7 CPs

Instruction parcel and following parcel both in a buffer and branch address not in a buffer, 18 CPs. Additional time is needed if a memory conflict exists. The time to resolve a memory conflict depends on factors present.

SPECIAL CASES: Instruction 0050*jk* executes as if it were a 2-parcel instruction. Even though the parcel following the first parcel of instruction 0050*jk* is not used, it can cause a delay of instruction 0050*jk* if it is out of buffer. See execution times above.

INSTRUCTION 006

CAL Syntax	Description	Octal Code
J *exp*	Branch to *ijkm*	006*ijkm*

The 2-parcel instruction 006 sets the P register to the parcel address specified by the low-order 24 bits of the *ijkm* field. Execution continues at that address. The high-order bit of the *ijkm* field is ignored.

HOLD ISSUE CONDITIONS:	Second parcel in different buffer, 2 CP delay Second parcel not in a buffer
EXECUTION TIME:	Instruction issue: Both parcels of instruction in the same buffer and branch address in a buffer, 5 CPs Both parcels of instruction in the same buffer and branch address not in a buffer, 16 CPs. Additional time is needed if a memory conflict exists. The time to resolve a memory conflict depends on factors present.
SPECIAL CASES:	None

INSTRUCTION 007

CAL Syntax	Description	Octal Code
R *exp*	Return jump to *ijkm*; set B00 to (P)+2.	007*ijkm*

The 2-parcel instruction 007 sets register B00 to the address of the parcel following the second parcel of the instruction. The P register is then set to the parcel address specified by the low-order 24 bits of the *ijkm* field. Execution continues at that address. The high-order bit of the *ijkm* field is ignored. This instruction provides a return linkage for subroutine calls. The subroutine is entered through a return jump. The subroutine can return to the caller at the instruction following the call by executing a branch to the contents of the B00 register.

HOLD ISSUE CONDITIONS:	Instruction 034 or 035 in process
	Second parcel in a different buffer, 2 CP delay
	Second parcel not in a buffer
EXECUTION TIME:	Instruction issue: Both parcels of instruction in the same buffer and branch address in a buffer, 5 CPs
	Both parcels of instruction in the same buffer and branch address not in a buffer, 16 CPs. Additional time is needed if a memory conflict exists. The time to resolve a memory conflict depends on factors present.
SPECIAL CASES:	None

CAL Syntax	Description	Octal Code
JAZ *exp*	Branch to *ijkm* if (A0)=0 (i_2=0)	010*ijkm*
JAN *exp*	Branch to *ijkm* if (A0)≠0 (i_2=0)	011*ijkm*
JAP *exp*	Branch to *ijkm* if (A0) positive, includes (A0)=0 (i_2=0)	012*ijkm*
JAM *exp*	Branch to *ijkm* if (A0) negative (i_2=0)	013*ijkm*

The 2-parcel instructions 010 through 013 test the contents of A0 for the condition specified by the *h* field. If the condition is satisfied, the P register is set to the parcel address specified by the low-order 24 bits of the *ijkm* field and execution continues at that address. The high-order bit of the *ijkm* field must be 0. If the condition is not satisfied, execution continues with the instruction following the branch instruction.

HOLD ISSUE CONDITIONS: A0 busy in any one of the previous 3 CPs

Second parcel in a different buffer, 2 CP delay

Second parcel not in a buffer

EXECUTION TIME: Instruction issue for branch taken:

Both parcels of instruction in the same buffer, branch taken, and branch address in a buffer, 5 CPs

Both parcels of instruction in the same buffer, branch taken, and branch address not in a buffer; 16 CPs. Additional time is needed if a memory conflict exists. The time to resolve a memory conflict is indeterminate.

Both parcels of instruction in different buffers, branch taken, and branch address in a buffer; 7 CPs.

Both parcels of instruction in different buffers, branch taken, and branch address not in a buffer; 18 CPs.

EXECUTION TIME: (continued)

Second parcel of instruction not in a buffer, branch taken, and branch address in a buffer; 18 CPs.

Second parcel of instruction not in a buffer, branch taken, and branch address not in buffer; 29 CPs.

Instruction issue for branch not taken:

Both parcels of instruction in the same buffer, branch not taken, and next instruction in the same instruction buffer, 2 CPs

Both parcels of instruction in the same buffer, branch not taken, and next instruction in different instruction buffer, 4 CPs

Both parcels of instruction in the same buffer and branch not taken with next instruction in memory; 16 CPs.

Both parcels of instruction in different buffers and branch not taken; 4 CPs.

Second parcel of instruction not in a buffer and branch not taken; 15 CPs.

NOTE

Whenever a fetch occurs, memory conflicts may produce a delay.

SPECIAL CASES:

(A0)=0 is considered a positive condition.

High-order bit of *i* designator (i_2) must be 0.

CAL Syntax		Description	Octal Code
JSZ	*exp*	Branch to *ijkm* if (S0)=0 (i_2=0)	014*ijkm*
JSN	*exp*	Branch to *ijkm* if (S0)≠0 (i_2=0)	015*ijkm*
JSP	*exp*	Branch to *ijkm* if (S0) positive, includes (S0)=0 (i_2=0)	016*ijkm*
JSM	*exp*	Branch to *ijkm* if (S0) negative (i_2=0)	017*ijkm*

The 2-parcel instructions 014 through 017 test the contents of S0 for the condition specified by the *h* field. If the condition is satisfied, the P register is set to the parcel address specified by the low-order 24 bits of the *ijkm* field and execution continues at that address. The high-order bit of the *ijkm* field must be 0. If the condition is not satisfied, execution continues with the instruction following the branch instruction.

HOLD ISSUE CONDITIONS: S0 busy in any one of the previous 3 CPs

Second parcel in a different buffer, 2 CP delay

Second parcel not in a buffer

EXECUTION TIME: Instruction issue for branch taken:

Both parcels of instruction in the same buffer, branch taken, and branch address in a buffer, 5 CPs

Both parcels of instruction in the same buffer, branch taken, and branch address not in a buffer; 16 CPs. Additional time is needed if a memory conflict exists. The time to resolve a memory conflict is indeterminate.

Both parcels of instruction in different buffers, branch taken, and branch address in a buffer; 7 CPs.

Both parcels of instruction in different buffers, branch taken, and branch address not in a buffer; 18 CPs.

EXECUTION TIME: (continued)

Second parcel of instruction not in a buffer, branch taken, and branch address in a buffer; 18 CPs.

Second parcel of instruction not in a buffer, branch taken, and branch address not in buffer; 29 CPs.

Instruction issue for branch not taken:

Both parcels of instruction in the same buffer, branch not taken, and next instruction in the same instruction buffer, 2 CPs

Both parcels of instruction in the same buffer, branch not taken, and next instruction in different instruction buffer, 4 CPs

Both parcels of instruction in the same buffer and branch not taken with next instruction in memory; 16 CPs.

Both parcels of instruction in different buffers and branch not taken; 4 CPs.

Second parcel of instruction not in a buffer and branch not taken; 15 CPs.

NOTE

Whenever a fetch occurs, memory conflicts may produce a delay.

SPECIAL CASES:

(S0)=0 is considered a positive condition.

High-order bit of *i* designator (i_2) must be 0.

INSTRUCTION 01*h*

CAL Syntax	Description	Octal Code
A*h* *exp*	Transmit *ijkm* to A*h* (i_2=1)	01*hijkm*

The 2-parcel instruction 01*h* enters a 24-bit value into A*h* that is composed of the low-order 24 bits of the *ijkm* field. The high-order bit of the *ijkm* field must be set to distinguish the 01*h* instruction from the 010-017 branches.

HOLD ISSUE CONDITIONS: A*h* reserved

Second parcel not in a buffer

Second parcel in a different buffer

EXECUTION TIME: Instruction issue:
Both parcels in same buffer, 2 CPs

Both parcels in different buffers, 4 CPs

A*h* ready, 1 CP

SPECIAL CASES: High-order bit of *i* designator (i_2) must be 1.

CAL Syntax	Description	Octal Code
A*i* *exp*	Transmit *jkm* to A*i*	020*ijkm*
A*i* *exp*	Transmit ones complement of *jkm* to A*i*	021*ijkm*

The 2-parcel instruction 020 enters a 24-bit value into A*i* composed of the 22-bit *jkm* field and 2 high-order bits of 0.

The 2-parcel instruction 021 enters a 24-bit value that is the complement of a value formed by the 22-bit *jkm* field and 2 high-order bits of 0 into A*i*. The complement is formed by changing all 1 bits to 0 and all 0 bits to 1. Thus, for instruction 021, the high-order 2 bits of A*i* are set to 1. The instruction provides a means of entering a negative value into A*i*. However, if the instruction is used to enter a negative number, the positive number used in the *jkm* field must be one smaller than the absolute value of the expected final negative number.

HOLD ISSUE CONDITIONS:	A*i* reserved
	Second parcel not in a buffer
EXECUTION TIME:	Instruction issue: Both parcels in same buffer, 2 CPs
	Both parcels in different buffers, 4 CPs
	A*i* ready, 1 CP
SPECIAL CASES:	None

INSTRUCTION 022

CAL Syntax	Description	Octal Code
A*i* *exp*	Transmit *jk* to A*i*	022*ijk*

Instruction 022 enters the 6-bit quantity from the *jk* field into the low-order 6 bits of A*i*. The high-order 18 bits of A*i* are zeroed. No sign extension occurs.

HOLD ISSUE CONDITIONS:	A*i* reserved
EXECUTION TIME:	Instruction issue, 1 CP
	A*i* ready, 1 CP
SPECIAL CASES:	None

INSTRUCTION 023

CAL Syntax	Description	Octal Code
A*i* S*j*	Transmit (S*j*) to A*i*	023*ij*0
A*i* VL	Read vector length	023*i*01

Instruction 023*ij*0 enters the low-order 24 bits of (S*j*) into A*i*. The high-order bits of (S*j*) are ignored.

Instruction 023*i*01 enters the content of the VL register into A*i*.

HOLD ISSUE CONDITIONS: A*i* reserved

For instruction 023*ij*0, S*j* reserved (except S0)

EXECUTION TIME: Instruction issue, 1 CP

A*i* ready, 1 CP

SPECIAL CASES: (S*j*)=0 if *j*=0.

If (A1)=0, the sequence:

```
  VL    A1
  A2    VL
```

leaves (A2)=100_8

If (A1)=23_8, the sequence:

```
  VL    A1
  A2    VL
```

leaves (A2)=23_8

If (A1)=123_8, the sequence:

```
  VL    A1
  A2    VL
```

leaves (A2)=23_8

The 2^6 bit in the VL is a 1 if the low-order 6 bits are 0; otherwise, the 2^6 bit is a 0.

CAL Syntax	Description	Octal Code
A*i* B*jk*	Transmit (B*jk*) to A*i*	024*ijk*
B*jk* A*i*	Transmit (A*i*) to B*jk*	025*ijk*

Instruction 024 enters the contents of B*jk* into A*i*.

Instruction 025 enters the contents of A*i* into B*jk*.

HOLD ISSUE CONDITIONS:	Instruction 034 or 035 in process
	For instruction 024*ijk*, instruction 025*ijk* issued in previous CP
	A*i* reserved
EXECUTION TIME:	For instruction 024, A*i* ready, 1 CP
	Instruction issue, 1 CP
SPECIAL CASES:	None

INSTRUCTION 026

CAL Syntax		Description	Octal Code
A*i*	PS*j*	Population count of (S*j*) to A*i*	026*ij*0
A*i*	QS*j*	Population count parity of (S*j*) to A*i*	026*ij*1
A*i*	SB*j*	Transfer (SB*j*) to A*i*	026*ij*7

Instruction 026*ij*0 counts the number of bits set to 1 in (S*j*) and enters the result into the low-order 7 bits of A*i*. The high-order 17 bits of A*i* are zeroed. If (S*j*)=0, then (A*i*)=0.

Instruction 026*ij*1 counts the number of bits set to 1 in (S*j*). Then, the low-order bit, showing the odd/even state of the result is transferred to the low-order bit position of the A*i* register. The high-order 23 bits are cleared. The actual population count is not transferred.

Instructions 026*ij*0 and 026*ij*1 are executed in the Population/ Leading Zero Count functional unit.

Instruction 026*ij*7 transfers the contents of the SB*j* register shared between the CPUs to A*i*.

HOLD ISSUE CONDITIONS:	A*i* reserved
	S*j* reserved (except S0)
	For instruction 026*ij*7, hold issue 1 CP, then 2+† CP more after A*i* not reserved. Minimum 3 CP hold.
EXECUTION TIME:	Instruction issue, 1 CP
	For instructions 026*ij*0 and 026*ij*1, A*i* ready 4 CPs
	For instruction 026*ij*7, A*i* ready 1 CP
SPECIAL CASES:	For instructions 026*ij*0 and 026*ij*1, (A*i*)=0 if *j*=0.
	For instruction 026*ij*7, (A*i*)=0 if CLN=0.

† If more than one CPU attempts to access semaphores or shared registers in the same clock period, a scanner will resolve the conflict. See shared register explanation in section 2.

INSTRUCTION 027

CAL Syntax	Description	Octal Code
A*i* ZS*j*	Leading zero count of (S*j*) to A*i*	027*ij*0
SB*j* A*i*	Transfer (A*i*) to SB*j*	027*ij*7

Instruction 027*ij*0 counts the number of leading zeros in S*j* and enters the result into the low-order 7 bits of A*i*. The high-order 17 bits of A*i* are zeroed. Instruction 027*ij*0 is executed in the Population/Leading Zero Count functional unit.

Instruction 027*ij*7 stores (A*i*) to the SB*j* register, which is shared between the CPUs in the same cluster.

HOLD ISSUE CONDITIONS:	For instruction 027*ij*0, instruction 033 issued in CP 2
	A*i* reserved
	S*j* reserved (except S0)
	For instruction 027*ij*7, hold issue 1 CP, then 2+† CP more after A*i* not reserved. Minimum 3 CP hold.
EXECUTION TIME:	Instruction issue, 1 CP
	For instruction 027*ij*0, A*i* ready, 3 CPs
	For instruction 027*ij*7, SB*j* ready 1 CP
SPECIAL CASES:	For instruction 027*ij*0, (A*i*)=64 if *j*=0.
	For instruction 027*ij*0, (A*i*)=0 if (S*j*) is negative.
	Instruction 027*ij*7 is a no-op if CLN=0.

† If more than one CPU attempts to access semaphores or shared registers in the same clock period, a scanner will resolve the conflict. See shared register explanation in section 2.

CAL Syntax	Description	Octal Code
A*i* A*j*+A*k*	Integer sum of (A*j*) and (A*k*) to A*i*	030*ijk*
A*i* A*k*†	Transmit (A*k*) to A*i*	030*i*0*k*
A*i* A*j*+1†	Integer sum of (A*j*) and 1 to A*i*	030*ij*0
A*i* A*j*-A*k*	Integer difference (A*j*) less (A*k*) to A*i*	031*ijk*
A*i* -1†	Transmit -1 to A*i*	031*i*00
A*i* -A*k*†	Transmit the negative of (A*k*) to A*i*	031*i*0*k*
A*i* A*j*-1†	Integer difference (A*j*) less 1 to A*i*	031*ij*0

Instruction 030 forms the integer sum of (A*j*) and (A*k*) and enters the result into A*i*. No overflow is detected.

Instruction 031 forms the integer difference of (A*j*) and (A*k*) and enters the result into A*i*. No overflow is detected.

Instructions 030 and 031 are executed in the Address Add functional unit.

HOLD ISSUE CONDITIONS:	A*i* reserved
	A*j* or A*k* reserved (except A0)
EXECUTION TIME:	Instruction issue, 1 CP
	A*i* ready, 2 CPs
SPECIAL CASES:	For instruction 030: (A*i*)=(A*k*) if *j*=0 and *k*≠0. (A*i*)=1 if *j*=0 and *k*=0. (A*i*)=(A*j*) + 1 if *j*≠0 and *k*=0.
	For instruction 031: (A*i*)= -(A*k*) if *j*=0 and *k*≠0. (A*i*)= -1 if *j*=0 and *k*=0. (A*i*)=(A*j*) - 1 if *j*≠0 and *k*=0.

† Special CAL syntax

INSTRUCTION 032

CAL Syntax	Description	Octal Code
A*i* A*j**A*k*	Integer product of (A*j*) and (A*k*) to A*i*	032*ijk*

Instruction 032 forms the integer product of (A*j*) and (A*k*) and enters the low-order 24 bits of the result into A*i*. No overflow is detected.

Instruction 032 is executed in the Address Multiply functional unit.

HOLD ISSUE CONDITIONS:	A*i* reserved
	A*j* or A*k* reserved (except A0)
EXECUTION TIME:	Instruction issue, 1 CP
	A*i* ready, 4 CPs
SPECIAL CASES:	(A*i*)=0 if *j*=0. (A*k*)=1 if *k*=0. Thus, (A*i*)=(A*j*) if *j*≠0 and *k*=0.

INSTRUCTION 033

CAL Syntax		Description	Octal Code
A*i*	CI	Channel number of highest priority interrupt request to A*i*	033*i*00
A*i*	CA,A*j*	Current address of channel (A*j*) to A*i*	033*ij*0
A*i*	CE,A*j*	Error flag of channel (A*j*) to A*i*	033*ij*1

Instruction 033 enters channel status information into A*i*. The *j* and *k* designators and the contents of A*j* define the desired information.

The channel number of the highest priority interrupt request is entered into A*i* when the *j* designator is 0. The contents of A*j* specify a channel number when the *j* designator is nonzero. The value of the Current Address (CA) register for the channel is entered into A*i* when the *k* designator is 0. The error flag for the channel is entered into the low-order bit of A*i* when the *k* designator is 1. The high-order bits of A*i* are cleared. The error flag can be cleared only in monitor mode using instruction 0012.

Instruction 033 does not interfere with channel operation and is not protected from user execution.

HOLD ISSUE CONDITIONS:	A*i* reserved
	A*j* reserved (except A0)
EXECUTION TIME:	Instruction issue, 1 CP
	A*i* ready, 4 CPs
SPECIAL CASES:	(A*i*)=Highest priority channel causing interrupt if (A*j*)=0.
	(A*i*)=Current address of channel (A*j*) if (A*j*)$\neq$0 and *k*=0.
	(A*i*)=I/O error flag of channel (A*j*) if (A*j*)$\neq$0 and *k*=1.

SPECIAL CASES: (continued)

6 CPs must elapse after instruction 0012*j*0 issues before issuing instruction 033*i*00.

Before the results of a 033*ij*0 instruction to channels 10-17 or a 033*ij*1 instruction to channels 6 or 7 are valid, there is a 12 CP latency. Therefore, before a 033*ij*X instruction can be issued to these channels 12 CPs must elapse after issuing a channel function or completing a channel transfer.

If instruction 033 issues every 10 CPs (in a loop), the same results will always be returned to A(*i*).

When *k*=1 and (A*j*)=6 or 7:

Bits 2^0 through 2^{17} contain the remaining block length.

Bit 2^{18} indicates a request in progress.

Bit 2^{19} will return a 0.

Bit 2^{20} indicates a block length error.

Bit 2^{21} indicates either an SSD double-bit memory error (during a read SSD operation) or an SSD double-bit channel error (during a write SSD operation).

Bit 2^{22} indicates a CPU double-bit memory error.

Bit 2^{23} indicates a fatal error (if bit 2^{20}, 2^{21}, or 2^{22} is set).

INSTRUCTIONS 034 - 037

CAL Syntax	Description	Octal Code
B*jk*,A*i* ,A0	Block transfer (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*	034*ijk*
B*jk*,A*i* 0,A0 †	Block transfer (A*i*) words from memory starting at address (A0) to B registers starting at register *jk*	034*ijk*
,A0 B*jk*,A*i*	Block transfer (A*i*) words from B registers starting at register *jk* to memory starting at address (A0)	035*ijk*
0,A0 B*jk*,A*i* †	Block transfer (A*i*) words from B registers starting at register *jk* to memory starting at address (A0)	035*ijk*
T*jk*,A*i* ,A0	Block transfer (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*	036*ijk*
T*jk*,A*i* 0,A0 †	Block transfer (A*i*) words from memory starting at address (A0) to T registers starting at register *jk*	036*ijk*
,A0 T*jk*,A*i*	Block transfer (A*i*) words from T registers starting at register *jk* to memory starting at address (A0)	037*ijk*
0,A0 T*jk*,A*i* †	Block transfer (A*i*) words from T registers starting at register *jk* to memory starting at address (A0)	037*ijk*

Instructions 034 through 037 perform block transfers between memory and B or T registers.

In all the instructions, the amount of data transferred is specified by the low-order 7 bits of (A*i*). See special cases for details.

The first register involved in the transfer is specified by *jk*. Successive transfers involve successive B or T registers until B77 or T77 is reached. Since processing of the registers is circular, B00 is processed after B77 and T00 is processed after T77 if the count in (A*i*) is not exhausted.

† Special CAL syntax

The first memory location referenced by the transfer instruction is specified by (A0). The A0 register contents are not altered by execution of the instruction. Memory references are incremented by 1 for successive transfers.

For transfers of B registers to memory, each 24-bit value is right adjusted in the word, high-order 40 bits are zeroed. When transferring from memory to B registers, only low-order 24 bits are transmitted; high-order 40 bits are ignored.

HOLD ISSUE CONDITIONS:	A0 reserved
	A*i* reserved
	Scalar reference in CP1, CP2, or CP3
	For instruction 034, Port A busy or instruction 035 in process or uni-directional memory mode and Port C busy
	For instruction 035, Port C busy or instruction 034 in process or uni-directional memory mode and Port A or Port B busy
	For instruction 036, Port B busy or instruction 037 in process or uni-directional memory mode and Port C busy
	For instruction 037, Port C busy or instruction 036 in process or uni-directional memory mode and Port A or Port B busy
EXECUTION TIME:	Instruction issue, 1 CP
	For instruction 034 or 036: B or T register reserved 16 CPs + (A*i*) if (A*i*)≠0; 6 CPs if (A*i*)=0. Port A or B busy for (A*i*) + 6 CPs if (A*i*)≠0; 4 CPs if (A*i*)=0.
	For instruction 035 or 037: B or T register reserved 5 CPs + (A*i*) if (A*i*)≠0; 4 CPs if (A*i*)=0. Port C busy for (A*i*) + 6 CPs if (A*i*)≠0; 4 CPs if (A*i*)=0.

SPECIAL CASES:

(A*i*)=0 causes a zero-block transfer.

(A*i*) in the range greater than 100_8 and less than 200_8 causes a wrap-around condition.

If (A*i*) is greater than 177_8, bits 2^7 through 2^{23} are truncated. The block length is equal to the value of 2^0 through 2^6.

NOTE

Instruction 034 uses Port A, instruction 035 uses Port C, instruction 036 uses Port B, and instruction 037 uses Port C.

CAL Syntax	Description	Octal Code
S*i* *exp*	Transmit *jkm* to S*i*	040*ijkm*
S*i* *exp*	Transmit complement of *jkm* to S*i*	041*ijkm*

The 2-parcel instructions 040 and 041 enter immediate values into an S register.

Instruction 040 enters a 64-bit value composed of the 22-bit *jkm* field and 42 high-order bits of 0 into S*i*.

Instruction 041 enters a 64-bit value that is the complement of a value formed by the 22-bit *jkm* field and 42 high-order bits of 0 into S*i*. The complement is formed by changing all 1 bits to 0 and all 0 bits to 1. Thus, for instruction 041, the high-order 42 bits of S*i* are set to 1's. The instruction provides for entering a negative value into S*i*. Since the register value is the ones complement of *jkm*, to get the twos complement *jkm* should be 0 to get -1, 1 to get -2, 3 to get -4, etc.

HOLD ISSUE CONDITIONS:	S*i* reserved
	Second parcel not in a buffer
EXECUTION TIME:	Instruction issue: Both parcels in same buffer, 2 CPs Both parcels in different buffers, 4 CPs S*i* ready, 1 CP
SPECIAL CASES:	None

CAL Syntax		Description	Octal Code
S*i*	<*exp*	Form *exp* bits of ones mask in S*i* from right; *jk* field gets 64-*exp*.	042*ijk*
S*i*	#>*exp*†	Form *exp* bits of zeros mask in S*i* from left; *jk* field gets *exp*.	042*ijk*
S*i*	1†	Enter 1 into S*i*	042*i*77
S*i*	-1†	Enter -1 into S*i*	042*i*00
S*i*	>*exp*	Form *exp* bits of ones mask in S*i* from left; *jk* field gets *exp*.	043*ijk*
S*i*	#<*exp*†	Form *exp* bits of zeros mask in S*i* from right; *jk* field gets 64-*exp*.	043*ijk*
S*i*	0†	Clear S*i*	043*i*00

Instruction 042 generates a mask of 64-*jk* ones from right to left in S*i*. For example, if *jk*=0, S*i* contains all 1 bits (integer value= -1) and if *jk*=77_8, S*i* contains zeros in all but the low-order bit (integer value=1).

Instruction 043 generates a mask of *jk* ones from left to right in S*i*. For example, if *jk*=0, S*i* contains all 0 bits (integer value=0) and if *jk*=77_8, S*i* contains ones in all but the low-order bit (integer value= -2).

Instructions 042 and 043 are executed in the Scalar Logical functional unit.

HOLD ISSUE CONDITIONS:	S*i* reserved
EXECUTION TIME:	Instruction issue, 1 CP
	S*i* ready, 1 CP
SPECIAL CASES:	None

† Special CAL syntax

CAL Syntax	Description	Octal Code
S*i* S*j*&S*k*	Logical product of (S*j*) and (S*k*) to S*i*	044*ijk*
S*i* S*j*&SB†	Sign bit of (S*j*) to S*i*	044*ij*0
S*i* SB&S*j*†	Sign bit of (S*j*) to S*i* (*j*≠0)	044*ij*0
S*i* #S*k*&S*j*	Logical product of (S*j*) and complement of (S*k*) to S*i*	045*ijk*
S*i* #SB&S*j*†	(S*j*) with sign bit cleared to S*i*	045*ij*0
S*i* S*j*\S*k*	Logical difference of (S*j*) and (S*k*) to S*i*	046*ijk*
S*i* S*j*\SB†	Toggle sign bit of (S*j*), then enter into S*i*	046*ij*0
S*i* SB\S*j*†	Toggle sign bit of (S*j*), then enter into S*i* (*j*≠0)	046*ij*0
S*i* #S*j*\S*k*	Logical equivalence of (S*k*) and (S*j*) to S*i*	047*ijk*
S*i* #S*k*†	Transmit ones complement of (S*k*) to S*i*	047*i*0*k*
S*i* #S*j*\SB†	Logical equivalence of (S*j*) and sign bit to S*i*	047*ij*0
S*i* #SB\S*j*†	Logical equivalence of (S*j*) and sign bit to S*i* (*j*≠0)	047*ij*0
S*i* #SB†	Enter ones complement of sign bit into S*i*	047*i*00
S*i* S*j*!S*i*&S*k*	Scalar merge	050*ijk*
S*i* S*j*!S*i*&SB†	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*	050*ij*0
S*i* S*j*!S*k*	Logical sum of (S*j*) and (S*k*) to S*i*	051*ijk*
S*i* S*k*†	Transmit (S*k*) to S*i*	051*i*0*k*
S*i* S*j*!SB†	Logical sum of (S*j*) and sign bit to S*i*	051*ij*0
S*i* SB!S*j*†	Logical sum of (S*j*) and sign bit to S*i* (*j*≠0)	051*ij*0
S*i* SB†	Enter sign bit into S*i*	051*i*00

† Special CAL syntax

NOTE

For instructions 044 through 051, SB with no register designator is the sign bit, not Shared Address register.

Instructions 044 through 051 are executed in the Scalar Logical functional unit.

Instruction 044 forms the logical product (AND) of (S*j*) and (S*k*) and enters the result into S*i*. Bits of S*i* are set to 1 when corresponding bits of (S*j*) and (S*k*) are 1 as in the following example:

```
(Sj) = 1 1 0 0
(Sk) = 1 0 1 0
       -------
(Si) = 1 0 0 0
```

(S*j*) is transmitted to S*i* if the *j* and *k* designators have the same nonzero value. S*i* is cleared if the *j* designator is 0. The sign bit of (S*j*) is transmitted to S*i* if the *j* designator is nonzero and the *k* designator is 0.

Instruction 045 forms the logical product (AND) of (S*j*) and the complement of (S*k*) and enters the result into S*i*. Bits of S*i* are set to 1 when corresponding bits of (S*j*) and the complement of (S*k*) are 1 as in the following example where (S*k*') = complement of (S*k*):

```
if (Sk) = 1 0 1 0

          (Sj)  = 1 1 0 0
          (Sk') = 0 1 0 1
                  -------
          (Si)  = 0 1 0 0
```

S*i* is cleared if the *j* and *k* designators have the same value or if the *j* designator is 0. (S*j*) with the sign bit cleared is transmitted to S*i* if the *j* designator is nonzero and the *k* designator is 0.

Instruction 046 forms the logical difference (exclusive OR) of (S*j*) and (S*k*) and enters the result into S*i*. Bits of S*i* are set to 1 when corresponding bits of (S*j*) and (S*k*) are different as in the following example:

```
(Sj) = 1 1 0 0
(Sk) = 1 0 1 0
       -------
(Si) = 0 1 1 0
```

S*i* is cleared if the *j* and *k* designators have the same nonzero value. (S*k*) is transmitted to S*i* if the *j* designator is 0 and the *k* designator is nonzero. The sign bit of (S*j*) is complemented and the result is transmitted to S*i* if the *j* designator is nonzero and the *k* designator is 0.

Instruction 047 forms the logical equivalence of (S*j*) and (S*k*) and enters the result into S*i*. Bits of S*i* are set to 1 when corresponding bits of (S*j*) and (S*k*) are the same as in the following example:

$$\begin{array}{rl}
(Sj) = & 1\ 1\ 0\ 0 \\
(Sk) = & \underline{1\ 0\ 1\ 0} \\
(Si) = & 1\ 0\ 0\ 1
\end{array}$$

S*i* is set to all ones if the *j* and *k* designators have the same nonzero value. The complement of (S*k*) is transmitted to S*i* if the *j* designator is 0 and the *k* designator is nonzero. All bits except the sign bit of (S*j*) are complemented and the result is transmitted to S*i* if the *j* designator is nonzero and the *k* designator is 0. The result is the complement produced by instruction 046.

Instruction 050 merges the contents of (S*j*) with (S*i*) depending on the ones mask in S*k*. The result is defined by the following Boolean equation where S*k*' is the complement of S*k* as illustrated:

$$(Si) = (Sj)(Sk) + (Si)(Sk')$$

if (S*k*) = 1 1 1 1 0 0 0 0

$$\begin{array}{rl}
(Sk') = & 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\
(Si) = & 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0 \\
(Sj) = & \underline{1\ 0\ 1\ 0\ 1\ 0\ 1\ 0} \\
(Si) = & 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0
\end{array}$$

Instruction 050 is intended for merging portions of 64-bit words into a composite word. Bits of S*i* are cleared when the corresponding bits of S*k* are 1 if the *j* designator is 0 and the *k* designator is nonzero. The sign bit of (S*j*) replaces the sign bit of S*i* if the *j* designator is nonzero and the *k* designator is 0. The sign bit of S*i* is cleared if the *j* and *k* designators are both 0.

Instruction 051 forms the logical sum (inclusive OR) of (S*j*) and (S*k*) and enters the result into S*i*. Bits of S*i* are set when 1 of the corresponding bits of (S*j*) and (S*k*) is set as in the following example:

$$\begin{array}{rl}
(Sj) = & 1\ 1\ 0\ 0 \\
(Sk) = & \underline{1\ 0\ 1\ 0} \\
(Si) = & 1\ 1\ 1\ 0
\end{array}$$

(S*j*) is transmitted to S*i* if the *j* and *k* designators have the same nonzero value. (S*k*) is transmitted to S*i* if the *j* designator is 0 and the *k* designator is nonzero. (S*j*) with the sign bit set to 1 is transmitted to S*i* if the *j* designator is nonzero and the *k* designator is 0. A ones mask consisting of only the sign bit is entered into S*i* if the *j* and *k* designators are both 0.

HOLD ISSUE CONDITIONS:	S*i* reserved
	S*j* or S*k* reserved (except S0)
EXECUTION TIME:	Instruction issue, 1 CP
	S*i* ready, 1 CP
SPECIAL CASES:	(S*j*)=0 if *j*=0.
	(S*k*)=2^{63} if *k*=0.

CAL Syntax	Description	Octal Code
S0 S*i*<*exp*	Shift (S*i*) left *exp*=*jk* places to S0	052*ijk*
S0 S*i*>*exp*	Shift (S*i*) right *exp*=64-*jk* places to S0	053*ijk*
S*i* S*i*<*exp*	Shift (S*i*) left *exp*=*jk* places to S*i*	054*ijk*
S*i* S*i*>*exp*	Shift (S*i*) right *exp*=64-*jk* places to S*i*	055*ijk*

Instructions 052 through 055 are executed in the Scalar Shift functional unit. They shift values in an S register by an amount specified by *jk*. All shifts are end off with zero fill.

Instruction 052 shifts (S*i*) left *jk* places and enters the result into S0. Shift range is 0 through 63 left.

Instruction 053 shifts (S*i*) right by 64-*jk* places and enters the result into S0. Shift range is 1 through 64 right.

Instruction 054 shifts (S*i*) left *jk* places and enters the result into S*i*. Shift range is 0 through 63 left.

Instruction 055 shifts (S*i*) right by 64-*jk* places and enters the result into S*i*. Shift range is 1 through 64 right.

HOLD ISSUE CONDITIONS:	Instruction 056, 057, 060, or 061 issued in previous CP
	S*i* reserved
	For instructions 052 and 053, S0 reserved
EXECUTION TIME:	Instruction issue, 1 CP
	For instructions 052 and 053, S0 ready, 2 CPs
	For instructions 054 and 055, S*i* ready, 2 CPs
SPECIAL CASES:	None

CAL Syntax	Description	Octal Code
S*i* S*i*,S*j*<A*k*	Shift (S*i*) and (S*j*) left by (A*k*) places to S*i*	056*ijk*
S*i* S*i*,S*j*<1†	Shift (S*i*) and (S*j*) left one place to S*i*	056*ij*0
S*i* S*i*<A*k*†	Shift (S*i*) left (A*k*) places to S*i*	056*i*0*k*
S*i* S*j*,S*i*>A*k*	Shift (S*j*) and (S*i*) right by (A*k*) places to S*i*	057*ijk*
S*i* S*j*,S*i*>1†	Shift (S*j*) and (S*i*) right one place to S*i*	057*ij*0
S*i* S*i*>A*k*†	Shift (S*i*) right (A*k*) places to S*i*	057*i*0*k*

Instructions 056 and 057 are executed in the Scalar Shift functional unit. They shift 128-bit values formed by logically joining two S registers. Shift counts are obtained from register A*k*. All shift counts, (A*k*), are considered positive and all 24 bits of (A*k*) are used for the shift count. A shift of one place occurs if the *k* designator is 0. If *j*=0, the shifts function as if the shifted value were 64 bits rather than 128 bits since the S*j* value used is 0.

The shifts are circular if the shift count does not exceed 64 and the *i* and *j* designators are equal and nonzero. For instructions 056 and 057, (S*j*) is unchanged, provided *i*≠*j*. For shifts greater than 64, the shift is end off with zero fill. If *i*=*j* and the shift is greater than 64, the shift is the same as if the respective instruction 054 or 055 was used with a shift count 64 less.

Instruction 056 performs left shifts of (S*i*) and (S*j*) with (S*i*) initially the most significant bits of the double register. The high-order 64 bits of the result are transmitted to S*i*. S*i* is cleared if the shift count exceeds 127. Instruction 056 produces the same result as instruction 054 if the shift count does not exceed 63 and the *j* designator is 0.

Instruction 057 performs right shifts of (S*j*) and (S*i*) with (S*j*) initially the most significant bits of the double register. The low-order 64 bits of the result are transmitted to S*i*. S*i* is cleared if the shift count exceeds 127. Instruction 057 produces the same result as instruction 055 if the shift count does not exceed 63 and the *j* designator is 0.

† Special CAL syntax

HOLD ISSUE CONDITIONS:	S*i* reserved
	S*j* or A*k* reserved (except S0 and/or A0)
EXECUTION TIME:	Instruction issue, 1 CP
	S*i* ready, 3 CPs
SPECIAL CASES:	(S*j*)=0 if *j*=0.
	(A*k*)=1 if *k*=0.
	Circular shift if $i=j\neq 0$ and A*k* greater than or equal to 0 and less than or equal to 64.

CAL Syntax		Description	Octal Code
S*i*	S*j*+S*k*	Integer sum of (S*j*) and (S*k*) to S*i*	060*ijk*
S*i*	S*j*-S*k*	Integer difference of (S*j*) and (S*k*) to S*i*	061*ijk*
S*i*	-S*k*†	Transmit negative of (S*k*) to S*i*	061*i*0*k*

Instruction 060 forms the integer sums of (S*j*) and (S*k*) and enters the result into S*i*. No overflow is detected.

Instruction 061 forms the integer difference of (S*j*) and (S*k*) and enters the result into S*i*. No overflow is detected.

Instructions 060 and 061 are executed in the Scalar Add functional unit.

HOLD ISSUE CONDITIONS:	S*i* reserved
	S*j* or S*k* reserved (except S0)
EXECUTION TIME:	S*i* ready, 3 CPs
	Instruction issue, 1 CP
SPECIAL CASES:	(S*i*)=2^{63} if *j*=0 and *k*=0.
	For instruction 060: (S*i*)=(S*k*) if *j*=0 and *k*≠0. (S*i*)=(S*j*) with 2^{63} complemented if *j*≠0 and *k*=0.
	For instruction 061: (S*i*)= -(S*k*) if *j*=0 and *k*≠0. (S*i*)=(S*j*) with 2^{63} complemented if *j*≠0 and *k*=0.

† Special CAL syntax

CAL Syntax		Description	Octal Code
S*i*	S*j*+FS*k*	Floating-point sum of (S*j*) and (S*k*) to S*i*	062*ijk*
S*i*	+FS*k*†	Normalize (S*k*) to S*i*	062*i*0*k*
S*i*	S*j*-FS*k*	Floating-point difference of (S*j*) and (S*k*) to S*i*	063*ijk*
S*i*	-FS*k*†	Transmit normalized negative of (S*k*) to S*i*	063*i*0*k*

Instructions 062 and 063 are performed in the Floating-point Add functional unit. Operands are assumed to be in floating-point format. The result is normalized even if the operands are not normalized.

Instruction 062 forms the sum of the floating-point quantities in S*j* and S*k* and enters the normalized result into S*i*.

Instruction 063 forms the difference of the floating-point quantities in S*j* and S*k* and enters the normalized result into S*i*.

Overflow conditions are described in section 4. For floating-point operands with the sign bit set (bit=1), zero exponent and zero coefficient are treated as 0 (that is, all 64 bits=0).††

HOLD ISSUE CONDITIONS:	S*i* reserved
	S*j* or S*k* reserved (except S0)
	Instructions 170 through 173 in process, unit busy (VL) + 4 CPs
EXECUTION TIME:	Instruction issue, 1 CP
	S*i* ready, 6 CPs

† Special CAL syntax

†† Considered -0. No floating-point unit generates a -0 except the Floating-point Multiply functional unit if one of the operands was a -0. Normally, -0 occurs in logical manipulations when a sign is attached to a number; that number can be 0.

SPECIAL CASES:

For instruction 062:
(S*i*)=(S*k*) normalized if (S*k*) exponent is valid, *j*=0 and *k*≠0.
(S*i*)=(S*j*) normalized if (S*j*) exponent is valid, *j*≠0 and *k*=0.

For instruction 063:
(S*i*)= -(S*k*) normalized if (S*k*) exponent is valid, *j*=0 and *k*≠0. Sign of (S*i*) is opposite that of (S*k*) if (S*k*)≠0.
(S*i*)=(S*j*) normalized if (S*j*) exponent is valid, *j*≠0 and *k*=0.

CAL Syntax		Description	Octal Code
S*i*	S*j**FS*k*	Floating-point product of (S*j*) and (S*k*) to S*i*	064*ijk*
S*i*	S*j**HS*k*	Half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*	065*ijk*
S*i*	S*j**RS*k*	Rounded floating-point product of (S*j*) and (S*k*) to S*i*	066*ijk*
S*i*	S*j**IS*k*	Reciprocal iteration; 2-(S*j*)*(S*k*) to S*i*	067*ijk*

Instructions 064 through 067 are executed in the Floating-point Multiply functional unit. Operands are assumed to be in floating-point format. The result is not guaranteed to be normalized if the operands are not normalized.

Instruction 064 forms the product of the floating-point quantities in S*j* and S*k* and enters the result into S*i*.

Instruction 065 forms the half-precision rounded product of the floating-point quantities in S*j* and S*k* and enters the result into S*i*. The low-order 19 bits of the result are cleared.

Instruction 066 forms the rounded product of the floating-point quantities in S*j* and S*k* and enters the result into S*i*.

Instruction 067 forms two minus the product of the floating-point quantities in S*j* and S*k* and enters the result into S*i*. This instruction is used in the divide sequence as described in section 4 under Floating-point Arithmetic.

In the evaluation C = 2-B*A, B must be a reciprocal of A of less than 47 significant bits and not the exact reciprocal; otherwise, C will be in error. The reciprocal produced by the reciprocal approximation instruction meets this criterion.

HOLD ISSUE CONDITIONS:	S*i* reserved
	S*j* or S*k* reserved (except S0)
	Instructions 160 through 167 in process, unit busy (VL) + 4 CPs
	Instructions 140 through 145 in process, Second Vector Logical unit busy (VL) + 4 CPs

EXECUTION TIME:	Instruction issue, 1 CP
	S*i* ready, 7 CPs
SPECIAL CASES:	(S*j*)=0 if *j*=0.
	(S*k*)=2^{63} if *k*=0.

If both exponent fields are 0, an integer multiply is performed. Correct integer multiply results are produced if the following conditions are met:

- Both operand sign bits are 0.
- The sum of the 0 bits to the right of the least significant 1 bit in the two operands is greater than or equal to 48.

The integer result obtained is the high-order 48 bits of the 96-bit product of the two operands.

INSTRUCTION 070

CAL Syntax	Description	Octal Code
S*i* /HS*j*	Floating-point reciprocal approximation of (S*j*) to S*i*	070*ij*0

Instruction 070 is executed in the Reciprocal Approximation functional unit.

Instruction 070 forms an approximation to the reciprocal of the normalized floating-point quantity in S*j* and enters the result into S*i*. This instruction occurs in the divide sequence to compute the quotient of two floating-point quantities as described in section 4 under Floating-point Arithmetic.

The reciprocal approximation instruction produces a result of 30 significant bits. The low-order 18 bits are zeros. The number of significant bits can be extended to 48 using the reciprocal iteration instruction and a multiply.

HOLD ISSUE CONDITIONS:	S*i* reserved
	S*j* reserved (except S0)
	Instruction 174 in process, unit busy (VL) + 4 CPs
EXECUTION TIME:	S*i* ready, 14 CPs
	Instruction issue, 1 CP
SPECIAL CASES:	(S*i*) is meaningless if (S*j*) is not normalized; the unit assumes that bit 2^{47} of (S*j*)=1; no test is made of this bit.
	(S*j*)=0 produces a range error; the result is meaningless.
	(S*j*)=0 if *j*=0.

INSTRUCTION 071

CAL Syntax	Description	Octal Code
S*i* A*k*	Transmit (A*k*) to S*i* with no sign extension	071*i*0*k*
S*i* +A*k*	Transmit (A*k*) to S*i* with sign extension	071*i*1*k*
S*i* +FA*k*	Transmit (A*k*) to S*i* as unnormalized floating-point number	071*i*2*k*
S*i* 0.6	Transmit constant 0.75 x 2^{48} to S*i*	071*i*30
S*i* 0.4	Transmit constant 0.5 to S*i*	071*i*40
S*i* 1.	Transmit constant 1.0 to S*i*	071*i*50
S*i* 2.	Transmit constant 2.0 to S*i*	071*i*60
S*i* 4.	Transmit constant 4.0 to S*i*	071*i*70

Instruction 071 performs functions that depend on the value of the *j* designator. The functions are concerned with transmitting information from an A register to an S register and with generating frequently used floating-point constants.

When the *j* designator is 0, the 24-bit value in A*k* is transmitted to S*i*. The value is treated as an unsigned integer. The high-order bits of S*i* are zeros.

When the *j* designator is 1, the 24-bit value in A*k* is transmitted to S*i*. The value is treated as a signed integer. The sign bit of A*k* is extended through the high-order bit of S*i*.

When the *j* designator is 2, the 24-bit value in A*k* is transmitted to S*i* as an unnormalized floating-point quantity (the result is then added to 0 to normalize). For this instruction, the exponent in bits 2^{62} through 2^{48} is set to 40060_8. The sign of the coefficient is set according to the sign of A*k*. If the sign bit of A*k* is set, the twos complement of A*k* is entered into S*i* as the magnitude of the coefficient and bit 2^{63} of S*i* is set for the sign of the coefficient.

A sequence of instructions is used to convert an integer whose absolute value is less than 24 bits to floating-point format:

```
    CAL code:  A1   S1
               S1   +FA1
               S1   +FS1      9 CPs required
```

When the *j* designator is 3, the floating-point constant of 0.75 x 2^{48} is entered into S*i* (0 40060 6000 0000 0000 0000_8). This constant is used to create floating-point numbers from integer numbers (positive and negative) whose absolute value is less than 47 bits. A sequence of instructions is used for conversion of an integer in S1:

```
    CAL code:   S2  0.6
                S1  S2-S1
                S1  S2-FS1     11 CPs required
```

When the *j* designator is 4, the floating-point constant 0.5 (= 0 40000 4000 0000 0000 0000_8) is entered into S*i*.

When the *j* designator is 5, the floating-point constant 1.0 (= 0 40001 4000 0000 0000 0000_8) is entered into S*i*.

When the *j* designator is 6, the floating-point constant 2.0 (= 0 40002 4000 0000 0000 0000_8) is entered into S*i*.

When the *j* designator is 7, the floating-point constant 4.0 (= 0 40003 4000 0000 0000 0000_8) is entered into S*i*.

HOLD ISSUE CONDITIONS:	S*i* reserved
	A*k* reserved (except A0); applies to all forms of the instruction, that is, *j* designators 0 through 7.
EXECUTION TIME:	Instruction issue, 1 CP
	S*i* ready, 2 CPs
SPECIAL CASES:	(A*k*)=1 if *k*=0.
	(S*i*)=(A*k*) if *j*=0.
	(S*i*)=(A*k*) sign extended if *j*=1.
	(S*i*)=(A*k*) unnormalized if *j*=2.
	(S*i*)=0.6 x 2^{60} (octal) if *j*=3.
	(S*i*)=0.4 x 2^{0} (octal) if *j*=4.
	(S*i*)=0.4 x 2^{1} (octal) if *j*=5.
	(S*i*)=0.4 x 2^{2} (octal) if *j*=6.
	(S*i*)=0.4 x 2^{3} (octal) if *j*=7.

CAL Syntax	Description	Octal Code
S*i* RT	Transmit (RTC) to S*i*	072*i*00
S*i* SM	Read semaphores to S*i*	072*i*02
S*i* ST*j*	Read (ST*j*) register to S*i*	072*ij*3
S*i* VM	Transmit (VM) to S*i*	073*i*00
†	Read performance counter into S*i*	073*i*11
†	Increment performance counter	073*i*21
†	Clear all maintenance modes	073*i*31
S*i* SR*j*	Transmit (SR*j*) to S*i*; *j*=0	073*ij*1
SM S*i*	Load semaphores from S*i*	073*i*02
ST*j* S*i*	Load (ST*j*) register from S*i*	073*ij*3
S*i* T*jk*	Transmit (T*jk*) to S*i*	074*ijk*
T*jk* S*i*	Transmit (S*i*) to T*jk*	075*ijk*

Instruction 072*i*00 enters the 64-bit value of the real-time clock (RTC) into S*i*. The clock is incremented by 1 each CP. The RTC can be set only by the monitor through use of instruction 0014*j*0.

Instruction 072*i*02 enters the values of all of the semaphores into S*i*. The 32-bit SM register is left justified in S*i* with SM00 occupying the sign bit.

Instruction 072*ij*3 enters the contents of ST*j* into S*i*.

Instruction 073*i*00 enters the 64-bit value of the VM register into S*i*. The VM register is usually read after being set by instruction 175.

Instruction 073*i*11 is used for performance monitoring and is privileged to monitor mode. Each execution of the 073*i*11 instruction advances a pointer and enters either the high-order or low-order bits of a performance counter into the high-order bits of S*i*. See Appendix C for information on performance monitoring.

† Not supported at this time

Instructions 073*i*21 and 073*i*31 are part of the SECDED maintenance mode functions and are executed only if the maintenance mode switch on the mainframe's control panel is on. Instruction 073*i*21 enables certain data bits to replace the 8 check bits used for SECDED as they are written into memory for any subsequent write to memory (except for I/O write to memory). Instruction 073*i*31 clears all three SECDED maintenance mode instructions: 001501, 001521, and 001531. See Appendix D for complete information on the SECDED maintenance modes.

Instruction 073*ij*1 enters the contents of the Status register SR*j* into S*i*. Instruction 073*i*01 returns the following status to the high-order bits of S*i*:

S*i* Bit	Description
2^{63}	Clustered, CLN ≠ 0 (CL)
2^{57}	Program state (PS)
2^{51}	Floating-point error occurred (FPS)
2^{50}	Floating-point interrupt enabled (IFP)
2^{49}	Operand range interrupt enabled (IOR)
2^{48}	Bidirectional memory enabled (BDM)
2^{41}†	Processor number bit 1 (PN1)
2^{40}†	Processor number bit 0 (PN0)
2^{34}†	Cluster number bit 2 (CLN2)
2^{33}†	Cluster number bit 1 (CLN1)
2^{32}†	Cluster number bit 0 (CLN0)

Instruction 073*i*02 sets the semaphores from 32 high-order bits of S*i*. SM00 receives the sign bit of S*i*.

Instruction 073*ij*3 enters the contents of S*i* into ST*j*.

Instruction 074 enters the contents of T*jk* into S*i*.

Instruction 075 enters the contents of S*i* into T*jk*.

HOLD ISSUE CONDITIONS: S*i* reserved

For instructions 074 and 075, instructions 036 through 037 in process

For instruction 074, instruction 075 issued in the previous CP

† These bit positions return a value of zero if not executed in monitor mode.

HOLD ISSUE CONDITIONS: (continued)	For instruction 073i00: Instruction 14x or 175 in process, VM busy for (VL) + 5 CPs Instruction 003 in process, VM busy for 1 CP For instructions 072ij3, 073ij3, and 073i02, hold issue 1 CP, then 2+† CP more after Si not reserved. Minimum 3 CP hold.
EXECUTION TIME:	Instruction issue, 1 CP All cases except 073ij3, result register ready, 1 CP For 073i02, SM ready, 1 CP
SPECIAL CASES:	For instructions 072i02 and 072ij3, (Si)=0 if CLN=0. Instructions 073i02 and 073ij3 are no-ops if CLN=0.

† If more than one CPU attempts to access semaphores or shared registers in the same clock period, a scanner will resolve the conflict. See shared register explanation in section 2.

CAL Syntax	Description	Octal Code
S*i* V*j*,A*k*	Transmit (V*j* element (A*k*)) to S*i*	076*ijk*
V*i*,A*k* S*j*	Transmit (S*j*) to V*i* element (A*k*)	077*ijk*
V*i*,A*k* 0†	Clear V*i* element (A*k*)	077*i*0*k*

Instructions 076 and 077 transmit a 64-bit quantity between a V register element and an S register.

Instruction 076 transmits the contents of an element of register V*j* to S*i*.

Instruction 077 transmits the contents of register S*j* to an element of register V*i*.

The low-order 6 bits of (A*k*) determine the vector element for either instruction.

HOLD ISSUE CONDITIONS:	A*k* reserved (except A0)
	For instruction 076, S*i* reserved or V*j* reserved as operand or as result
	For instruction 077, V*i* reserved as operand or as result or S*j* reserved
EXECUTION TIME:	Instruction issue, 1 CP
	For instruction 076, S*i* ready, 4 CPs
	For instruction 077, V*i* ready, 1 CP
SPECIAL CASES:	(S*j*)=0 if *j*=0.
	(A*k*)=1 if *k*=0.

† Special CAL syntax

INSTRUCTIONS 10*h* - 13*h*

CAL Syntax	Description	Octal Code
A*i* *exp*,A*h*	Read from ((A*h*) + *jkm*) to A*i*	10*hijkm*
A*i* *exp*,0†	Read from (*jkm*) to A*i*	100*ijkm*
A*i* *exp*,†	Read from (*jkm*) to A*i*	100*ijkm*
A*i* ,A*h*†	Read from (A*h*) to A*i*	10*hi*00 0
exp,A*h* A*i*	Store (A*i*) to (A*h*) + *jkm*	11*hijkm*
exp,0 A*i*†	Store (A*i*) to *jkm*	110*ijkm*
exp, A*i*†	Store (A*i*) to *exp*	110*ijkm*
,A*h* A*i*†	Store (A*i*) to (A*h*)	11*hi*00 0
S*i* *exp*,A*h*	Read from ((A*h*) + *jkm*) to S*i*	12*hijkm*
S*i* *exp*,0†	Read from (*exp*) to S*i*	120*ijkm*
S*i* *exp*,†	Read from (*exp*) to S*i*	120*ijkm*
S*i* ,A*h*†	Read from (A*h*) to S*i*	12*hi*00 0
exp,A*h* S*i*	Store (S*i*) to (A*h*) + *jkm*	13*hijkm*
exp,0 S*i*†	Store (S*i*) to *exp*	130*ijkm*
exp, S*i*†	Store (S*i*) to *exp*	130*ijkm*
,A*h* S*i*†	Store (S*i*) to (A*h*)	13*hi*00 0

The 2-parcel instructions 10*h* through 13*h* transmit data between memory and an A register or an S register.

If the enhanced addressing mode bit in the Exchange Package is not set, the content of A*h* (treated as a 22-bit signed integer) is added to the signed 22-bit integer in the *jkm* field to determine the memory address. Data base address bits 2^{22} and 2^{23} will determine which 4 million words of memory will be used. If the enhanced addressing mode bit (EAM) of the Exchange Package is set, the content of A*h* (treated as a 24-bit integer) is added to the sign extended 24-bit integer in the *jkm* field to determine the memory address.

† Special CAL syntax

If *h* is 0, (A*h*) is 0 and only the *jkm* field is used for the address. The address arithmetic is performed by an address adder similar to but separate from the Address Add functional unit.

Instructions 10*h* and 11*h* transmit 24-bit quantities to or from A registers. When transmitting data from memory to an A register, the high-order 40 bits of the memory word are ignored. On a store from A*i* into memory, the high-order 40 bits of the memory word are zeroed.

Instructions 12*h* and 13*h* transmit 64-bit quantities to or from register S*i*.

HOLD ISSUE CONDITIONS: Port A, B, or C busy

A*h* reserved or busy previous CP

For instructions 10*h* and 11*h*, A*i* reserved

For instructions 12*h* and 13*h*, S*i* reserved

Instructions 10*x* through 13*x* in CP 2 and CP 3 and conflict

Second parcel not in a buffer

Second parcel in different buffer, 2 CP

EXECUTION TIME: Instruction issue:
- Both parcels in same buffer, 2 CPs
- For instruction 10*h*, A*i* ready, 14 CPs
- For instruction 12*h*, S*i* ready, 14 CPs
- Bank ready for next scalar read or store, 4 CPs

NOTE

After issuing instructions 10*h* through 13*h*, attempting to issue instructions 034 through 037, 176, or 177 causes Ports A, B, or C to be considered busy for 4 CPs (plus additional CPs if there are conflicts).

SPECIAL CASES: If the enhanced addressing mode bit (EAM) of the Exchange Package is set, the *jkm* field is sign-extended to 24 bits.

CAL Syntax		Description	Octal Code
V*i*	S*j*&V*k*	Logical products of (S*j*) and (V*k* elements) to V*i* elements	140*ijk*
V*i*	V*j*&V*k*	Logical products of (V*j* elements) and (V*k* elements) to V*i* elements	141*ijk*
V*i*	S*j*!V*k*	Logical sums of (S*j*) and (V*k* elements) to V*i* elements	142*ijk*
V*i*	V*k*†	Transmit (V*k* elements) to V*i* elements	142*i*0*k*
V*i*	V*j*!V*k*	Logical sums of (V*j* elements) and (V*k* elements) to V*i* elements	143*ijk*
V*i*	S*j*\V*k*	Logical differences of (S*j*) and (V*k* elements) to V*i* elements	144*ijk*
V*i*	V*j*\V*k*	Logical differences of (V*j* elements) and (V*k* elements) to V*i* elements	145*ijk*
V*i*	0†	Clear V*i* elements	145*iii*
V*i*	S*j*!V*k*&VM	If VM bit=1, transmit (S*j*) to the corresponding element in V*i* If VM bit=0, transmit the (corresponding V*k* element) to the (corresponding V*i* element)	146*ijk*
V*i*	#VM&V*k*†	If VM bit=1, transmit (0) to the corresponding element in V*i* If VM bit=0, transmit the (corresponding V*k* element) to the (corresponding V*i* element)	146*i*0*k*
V*i*	V*j*!V*k*&VM	If VM bit=1, transmit the (corresponding V*j* element) to the (corresponding V*i* element) If VM bit=0, transmit the (corresponding V*k* element) to the (corresponding V*i* element)	147*ijk*

Instructions 140 through 145 can be executed in either the Full Vector Logical or the Second Vector Logical functional units, provided the Second Vector Logical Unit is enabled. If the Second Vector Logical unit is disabled, instructions 140 through 145 can be executed only in the Full Vector Logical unit. Instructions 146 and 147 execute in the

† Special CAL syntax

Full Vector Logical unit only. The number of operations performed is determined by the contents of the VL register. All operations start with element 0 of the V*i*, V*j*, or V*k* register and increment the element number by 1 for each operation performed. All results are delivered to V*i*.

For instructions 140, 142, 144, and 146, a copy of the content of S*j* is delivered to the functional unit. The copy of the content is held as one of the operands until completion of the operation. Therefore, S*j* can be changed immediately without affecting the vector operation. For instructions 141, 143, 145, and 147, all operands are obtained from V registers.

Instructions 140 and 141 form the logical products (AND) of operand pairs and enter the result into V*i*. Bits of an element of V*i* are set to 1 when the corresponding bits of (S*j*) or (V*j* element) and (V*k* element) are 1 as in the following:

```
(Sj) or (Vj element) = 1 1 0 0
(Vk element)         = 1 0 1 0
(Vi element)         = 1 0 0 0
```

Instructions 142 and 143 form the logical sums (inclusive OR) of operand pairs and deliver the results to V*i*. Bits of an element of V*i* are set to 1 when one of the corresponding bits of (S*j*) or (V*j* element) and (V*k* element) is 1 as in the following:

```
(Sj) or (Vj element) = 1 1 0 0
(Vk element)         = 1 0 1 0
(Vi element)         = 1 1 1 0
```

Instructions 144 and 145 form the logical differences (exclusive OR) of operand pairs and deliver the results of V*i*. Bits of an element are set to 1 when the corresponding bit of (S*j*) or (V*j* element) is different from (V*k* element) as in the following:

```
(Sj) or (Vj element) = 1 1 0 0
(Vk element)         = 1 0 1 0
(Vi element)         = 0 1 1 0
```

Instructions 146 and 147 transmit operands to V*i* depending on the contents of the VM register. Bit 2^{63} of the mask corresponds to element 0 of a V register. Bit 2^0 corresponds to element 63. Operand pairs used for the selection depend on the instruction. For instruction 146, the first operand is always (S*j*), the second operand is (V*k* element). For instruction 147, the first operand is (V*j* element) and the second operand is (V*k* element). If bit *n* of the vector mask is 1, the first operand is transmitted; if bit *n* of the mask is 0, the second operand, (V*k* element), is selected.

Examples:

1. If instruction 146 is to be executed and the following register conditions exist:

```
(VL)   = 4
(VM)   = 0 60000 0000 0000 0000 0000
(S2)   = -1
(V600) = 1
(V601) = 2
(V602) = 3
(V603) = 4
```

   Instruction 146726 is executed. Following execution, the first four elements of V7 contain the following values:

```
(V700) = 1
(V701) = -1
(V702) = -1
(V703) = 4
```

   The remaining elements of V7 are unaltered.

2. If instruction 147 is to be executed and the following register conditions exist:

```
(VL)   = 4
(VM)   = 0 600000 0000 0000 0000 0000
(V200) = 1     (V300)  = -1
(V201) = 2     (V301)  = -2
(V202) = 3     (V302)  = -3
(V203) = 4     (V303)  = -4
```

   Instruction 147123 is executed. Following execution, the first four elements of V1 contain the following values:

```
(V100) = -1
(V101) = 2
(V102) = 3
(V103) = -4
```

   The remaining elements of V1 are unaltered.

HOLD ISSUE CONDITIONS: V*k* reserved as operand

V*i* reserved as operand or result

For instructions 140, 142, 144, and 146, S*j* reserved

HOLD ISSUE CONDITIONS: (continued)	For instructions 141, 143, 145, and 147, V*j* reserved as operand
	For instructions 146 and 147, or instructions 140 through 145 with Second Vector Logical unit disabled: Instruction 14*x* or 175 in process, Full Vector Logical unit busy (VL) + 4 CPs
	For instructions 140 through 145 with Second Vector Logical unit enabled: See discussion of Second Vector Logical issue in section 4.
	Instruction 140 through 145 or 16*x* in progress in Second Vector Logical/Floating-point Multiply unit, Second Vector Logical unit busy (VL) + 4 CPs
	Instruction 140 through 147 or 175 in progress in Full Vector Logical unit, Full Vector Logical unit busy (VL) + 4 CPs
EXECUTION TIME:	Instruction issue, 1 CP
	V*j* or V*k* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 7 CPs if data available† for the Full Vector Logical unit; 9 CPs if available for the Second Vector Logical unit.
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	(S*j*)=0 if *j*=0.

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

CAL Syntax		Description	Octal Code
V*i*	V*j*<A*k*	Shift (V*j*) elements left by (A*k*) places to V*i* elements	150*ijk*
V*i*	V*j*<1[†]	Shift (V*j*) elements left one place to V*i* elements	150*ij*0
V*i*	V*j*>A*k*	Shift (V*j*) elements right by (A*k*) places to V*i* elements	151*ijk*
V*i*	V*j*>1[†]	Shift (V*j*) elements right one place to V*i* elements	151*ij*0

Instructions 150 and 151 are executed in the Vector Shift functional unit. The number of operations performed is determined by the contents of the VL register. Operations start with element 0 of the V*i* and V*j* registers and end with elements specified by (VL)-1.

All shifts are end off with zero fill. The shift count is obtained from (A*k*) and all 24 bits of A*k* are used for the shift count. Elements of V*i* are cleared if the shift count exceeds 63. All shift counts (A*k*) are considered positive.

Unlike shift instructions 052 through 055, these instructions receive the shift count from A*k*, rather than the *jk* fields.

HOLD ISSUE CONDITIONS: V*j* reserved as operand

V*i* reserved as operand or result

A*k* reserved (except A0)

Instructions 150 through 153 in process, unit busy (VL) + 4 CPs[††]

† Special CAL syntax

†† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

EXECUTION TIME:	V*j* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 8 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	(A*k*)=1 if *k*=0.

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

CAL Syntax	Description	Octal Code
V*i* V*j*,V*j*<A*k*	Double shifts of (V*j* elements) left (A*k*) places to V*i* elements	152*ijk*
V*i* V*j*,V*j*<1[†]	Double shifts of (V*j* elements) left one place to V*i* elements	152*ij*0
V*i* V*j*,V*j*>A*k*	Double shifts of (V*j* elements) right (A*k*) places to V*i* elements	153*ijk*
V*i* V*j*,V*j*>1[†]	Double shifts of (V*j* elements) right one place to V*i* elements	153*ij*0

Instructions 152 and 153 are executed in the Vector Shift functional unit. The instructions shift 128-bit values formed by logically joining the contents of two elements of the V*j* register. The direction of the shift determines whether the high-order bits or the low-order bits of the result are sent to V*i*. Shift counts are obtained from register A*k*.

All shifts are end off with zero fill.

The number of operations is determined by the contents of the VL register.

Instruction 152 performs left shifts. The operation starts with element 0 of V*j*. If (VL) is 1, element 0 is joined with 64 bits of 0, and the resulting 128-bit quantity is then shifted left by the amount specified by (A*k*). Only the one operation is performed. The 64 high-order bits remaining are transmitted to element 0 of V*i*.

If (VL) is 2, the operation starts with element 0 of V*j* being joined with element 1, and the resulting 128-bit quantity is then shifted left by the amount specified by (A*k*). The high-order 64 bits remaining are transmitted to element 0 of V*i*. Figure 5-7 illustrates this operation.

If (VL) is greater than 2, the operation continues by joining element 1 with element 2 and transmitting the 64-bit result to element 1 of V*i*. Figure 5-8 illustrates this operation.

If (VL) is 2, element 1 is joined with 64 bits of 0 and only two operations are performed. In general, the last element of V*j* as determined by (VL) is joined with 64 bits of zeros. Figure 5-9 illustrates this operation.

† Special CAL syntax

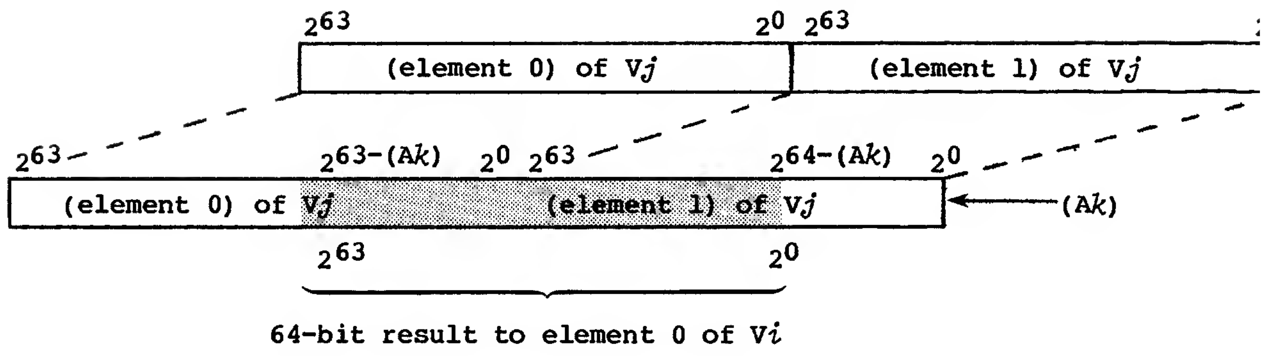


Figure 5-7. Vector left double shift, first element, VL greater than 1

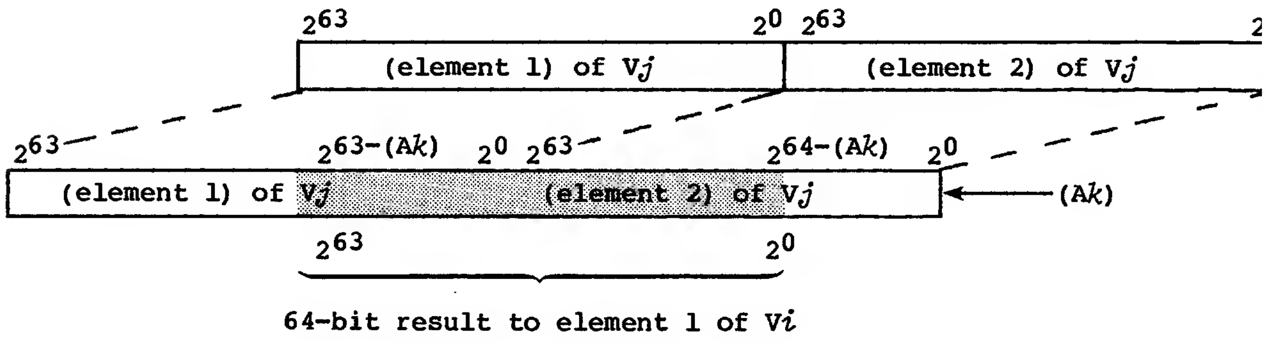


Figure 5-8. Vector left double shift, second element, VL greater than 2

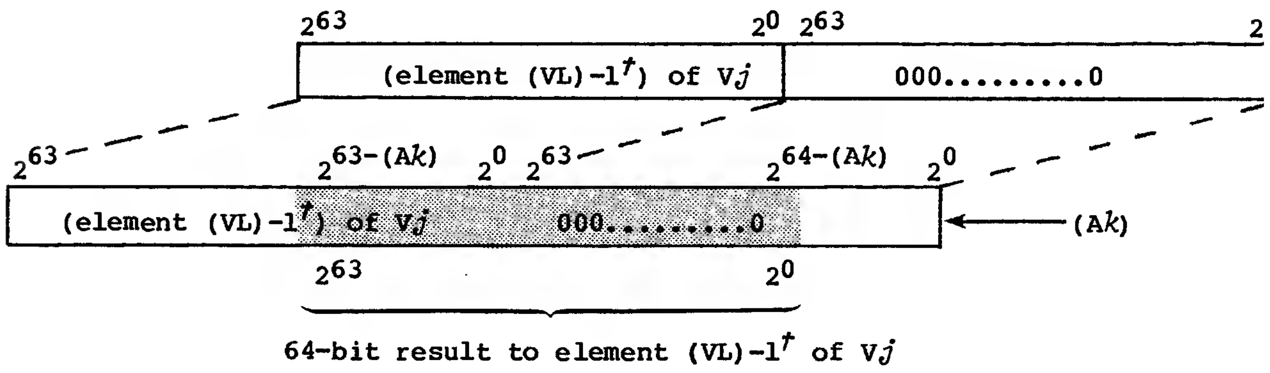


Figure 5-9. Vector left double shift, last element

† Elements are numbered 0 through 63 in the V registers; therefore, element (VL)-1 refers to the VLth element.

If (A*k*) is greater than or equal to 128, the result is all zeros. If (A*k*) is greater than 64, the result register contains at least (A*k*) - 64 zeros.

Examples:

1. If instruction 152 is to be executed and the following register conditions exist:

```
(VL)    = 4
(A1)    = 3
(V400) = 0 00000 0000 0000 0000 0007
(V401) = 0 60000 0000 0000 0000 0005
(V402) = 1 00000 0000 0000 0000 0006
(V403) = 1 60000 0000 0000 0000 0007
```

   Instruction 152541 is executed. Following execution, the first four elements of V5 contain the following values:

```
(V500) = 0 00000 0000 0000 0000 0073
(V501) = 0 00000 0000 0000 0000 0054
(V502) = 0 00000 0000 0000 0000 0067
(V503) = 0 00000 0000 0000 0000 0070
```

   Instruction 153 performs right shifts. The original element 0 of V*j* is joined with 64 high-order bits of 0 and the 128-bit quantity is shifted right by the amount specified by (A*k*). The 64 low-order bits of the result are transmitted to element 0 of V*i*. Figure 5-10 illustrates this operation.

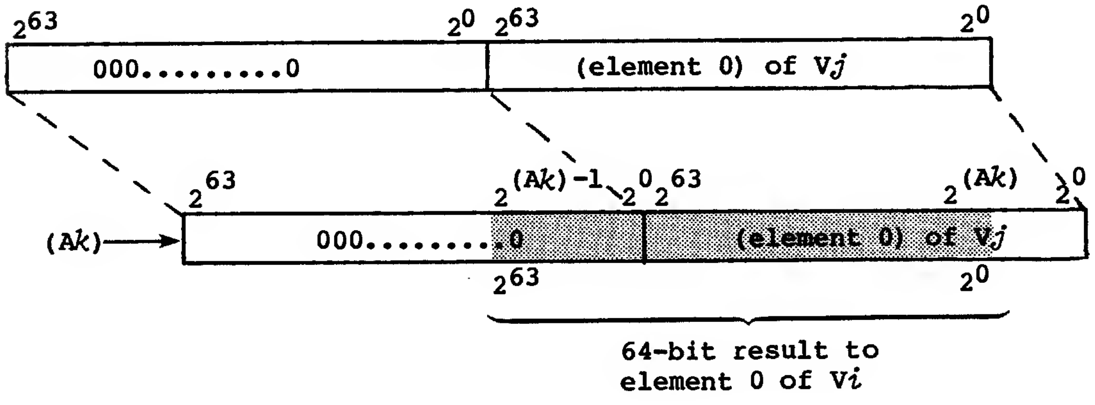


Figure 5-10. Vector right double shift, first element

If (VL)=1, only one operation is performed. In general, however, instruction execution continues by joining element 0 with element 1, shifting the 128-bit quantity by the amount specified by (A*k*), and transmitting the result to element 1 of V*i*. This operation is shown in figure 5-11.

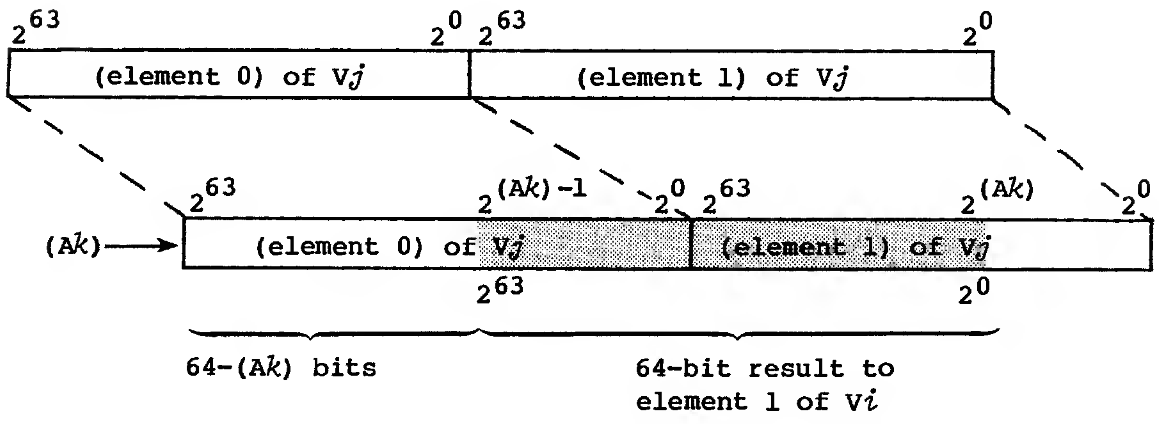


Figure 5-11. Vector right double shift, second element, VL greater than 1

The last operation performed by the instruction joins the last element of V*j* as determined by (VL) with the preceding element. Figure 5-12 illustrates this operation.

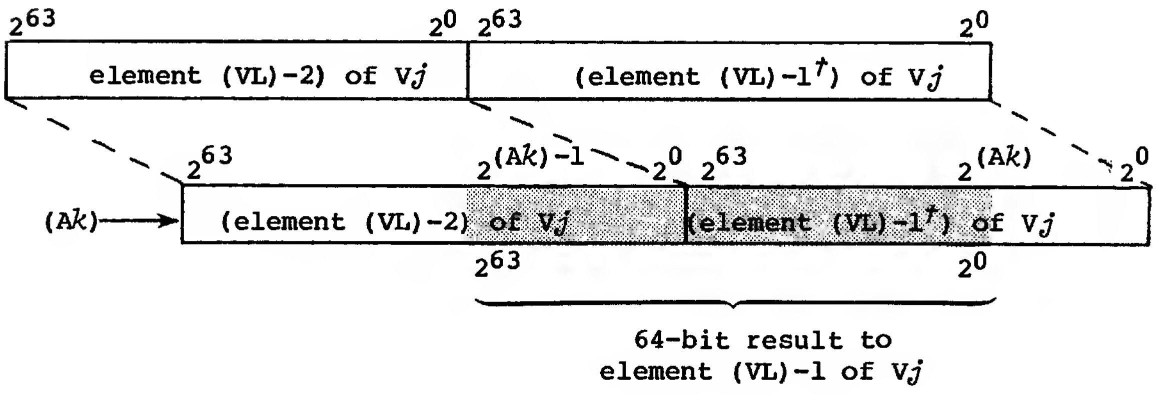


Figure 5-12. Vector right double shift, last operation

2. If an instruction 153 is to be executed and the following register conditions exist:

† Elements are numbered 0 through 63 in the V registers; therefore, element (VL)-1 refers to the VLth element.

```
(VL)   = 4
(A6)   = 3
(V200) = 0 00000 0000 0000 0000 0017
(V201) = 0 60000 0000 0000 0000 0006
(V202) = 1 00000 0000 0000 0000 0006
(V203) = 1 60000 0000 0000 0000 0007
```

Instruction 153026 is executed and following execution, register V0 contains the following values:

```
(V000) = 0 00000 0000 0000 0000 0001
(V001) = 1 66000 0000 0000 0000 0000
(V002) = 1 50000 0000 0000 0000 0000
(V003) = 1 56000 0000 0000 0000 0000
```

The remaining elements of V0 are unaltered.

HOLD ISSUE CONDITIONS:	V*j* reserved as operand
	V*i* reserved as operand or result
	A*k* reserved (except A0)
	Instructions 150 through 153 in process, unit busy (VL) + 4 CPs†
EXECUTION TIME:	Instruction issue, 1 CP
	V*j* ready in (VL) + 3 CPs if data available†
	For instruction 152, V*i* ready in (VL) + 9 CPs if data available†
	Instruction 153, V*i* ready in (VL) + 8 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	(A*k*)=1 if *k*=0.

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

CAL Syntax		Description	Octal Code
V*i*	S*j*+V*k*	Integer sums of (S*j*) and (V*k* elements) to V*i* elements	154*ijk*
V*i*	V*j*+V*k*	Integer sums of (V*j* elements) and (V*k* elements) to V*i* elements	155*ijk*
V*i*	S*j*-V*k*	Integer differences of (S*j*) and (V*k* elements) to V*i* elements	156*ijk*
V*i*	-V*k*†	Transmit negative of (V*k* elements) to V*i* elements	156*i*0*k*
V*i*	V*j*-V*k*	Integer differences of (V*j* elements) and (V*k* elements) to V*i* elements	157*ijk*

Instructions 154 through 157 are executed in the Vector Add functional unit.

Instructions 154 and 155 perform integer addition. Instructions 156 and 157 perform integer subtraction. The number of additions or subtractions performed is determined by the contents of the VL register. All operations start with element 0 of the V registers and increment the element number by 1 for each operation performed. All results are delivered to elements of V*i*. No overflow is detected.

Instructions 154 and 156 deliver a copy of (S*j*) to the functional unit where the copy is retained as one of the operands until the vector operation completes. The other operand is an element of V*k*. For instructions 155 and 157, both operands are obtained from V registers.

HOLD ISSUE CONDITIONS:	V*k* reserved as operand
	V*i* reserved as operand or result
	Instructions 154 through 157 in process, unit busy (VL) + 4 CPs†
	For instructions 154 and 156, S*j* reserved (except S0)
	For instructions 155 and 157, V*j* reserved as operand

† Special CAL syntax

EXECUTION TIME:	Instruction issue, 1 CP
	V*j* or V*k* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 8 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	For instruction 154, if *j*=0, then (S*j*)=0 and (V*i* element) = (V*k* element).
	For instruction 156, if *j*=0, then (S*j*)=0 and (V*i* element)= -(V*k* element).

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

CAL Syntax		Description	Octal Code
V*i*	S*j**FV*k*	Floating-point products of (S*j*) and (V*k* elements) to V*i* elements	160*ijk*
V*i*	V*j**FV*k*	Floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements	161*ijk*
V*i*	S*j**HV*k*	Half-precision rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements	162*ijk*
V*i*	V*j**HV*k*	Half-precision rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements	163*ijk*
V*i*	S*j**RV*k*	Rounded floating-point products of (S*j*) and (V*k* elements) to V*i* elements	164*ijk*
V*i*	V*j**RV*k*	Rounded floating-point products of (V*j* elements) and (V*k* elements) to V*i* elements	165*ijk*
V*i*	S*j**IV*k*	Reciprocal iterations; 2-(S*j*)*(V*k* elements) to V*i* elements	166*ijk*
V*i*	V*j**IV*k*	Reciprocal iterations; 2-(V*j* elements)* (V*k* elements) to V*i* elements	167*ijk*

Instructions 160 through 167 are executed in the Floating-point Multiply functional unit. The number of operations performed by an instruction is determined by the contents of the VL register. All operations start with element 0 of the V registers and increment the element number by 1 for each successive operation.

Operands are assumed to be in floating-point format. Instructions 160, 162, 164, and 166 deliver a copy of (S*j*) to the functional unit where the copy is retained as one of the operands until the completion of the operation. Therefore, S*j* can be changed immediately without affecting the vector operation. The other operand is an element of V*k*. For instructions 161, 163, 165, and 167, both operands are obtained from V registers.

All results are delivered to elements of V*i*. If either operand is not normalized, there is no guarantee that the products will be normalized. If neither operand is normalized, the product will not be normalized.

Out-of-range conditions are described in section 4.

Instruction 160 forms the products of the floating-point quantity in S*j* and the floating-point quantities in elements of V*k* and enters the results into V*i*.

Instruction 161 forms the products of the floating-point quantities in elements of V*j* and V*k* and enters the results into V*i*.

Instruction 162 forms the half-precision rounded products of the floating-point quantity in S*j* and the floating-point quantities in elements of V*k* and enters the results into V*i*. The low-order 19 bits of the result elements are zeroed.

Instruction 163 forms the half-precision rounded products of the floating-point quantities in elements of V*j* and V*k* and enters the results into V*i*. The low-order 19 bits of the result elements are zeroed.

Instruction 164 forms the rounded products of the floating-point quantity in S*j* and the floating-point quantities in elements of V*k* and enters the results into V*i*.

Instruction 165 forms the rounded products of the floating-point quantities in elements of V*j* and V*k* and enters the results into V*i*.

Instruction 166 forms for each element, two minus the product of the floating-point quantity in S*j* and the floating-point quantity in elements of V*k*. It then enters the results into V*i*. See the description of instruction 067 for more details.

Instruction 167 forms for each element pair, two minus the product of the floating-point quantities in elements of V*j* and V*k* and enters the results into V*i*. See the description of instruction 067 for more details.

HOLD ISSUE CONDITIONS:	V*k* reserved as operand
	V*i* reserved as operand or result
	Instruction 16*x* in process, unit busy (VL) + 4 CPs†

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

HOLD CONDITIONS: (continued)	Instructions 140-145 in process in Second Vector Logical unit. Unit busy (VL) + 4 CPs
	For instructions 160, 162, 164, and 166, S*j* reserved (except S0)
	For instructions 161, 163, 165, and 167, V*j* reserved as operand
EXECUTION TIME:	Instruction issue, 1 CP
	V*j* and V*k* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 12 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	(S*j*)=0 if *j*=0.

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

CAL Syntax		Description	Octal Code
V*i*	S*j*+FV*k*	Floating-point sums of (S*j*) and (V*k* elements) to V*i* element	170*ijk*
V*i*	+FV*k*†	Transmit normalized (V*k* elements) to V*i* elements	170*i*0*k*
V*i*	V*j*+FV*k*	Floating-point sums of (V*j* elements) and (V*k* elements) to V*i* elements	171*ijk*
V*i*	S*j*-FV*k*	Floating-point differences of (S*j*) and (V*k* elements) to V*i* elements	172*ijk*
V*i*	-FV*k*†	Transmit normalized negatives of (V*k* elements) to V*i* elements	172*i*0*k*
V*i*	V*j*-FV*k*	Floating-point differences of (V*j* elements) and (V*k* elements) to V*i* elements	173*ijk*

Instructions 170 through 173 are executed in the Floating-point Add functional unit. Instructions 170 and 171 perform floating-point addition; instructions 172 and 173 perform floating-point subtraction. The number of additions or subtractions performed by an instruction is determined by contents of the VL register. All operations start with element 0 of the V registers and increment the element number by 1 for each operation performed. All results are delivered to V*i* normalized and results are normalized even if the operands are not normalized.

Instructions 170 and 172 deliver a copy of (S*j*) to the functional unit where it remains as one of the operands until the completion of the operation. The other operand is an element of V*k*. For instructions 171 and 173, both operands are obtained from V registers. Out-of-range conditions are described in section 4.

HOLD ISSUE CONDITIONS:	V*k* reserved as operand
	V*i* reserved as operand or result

† Special CAL syntax

HOLD ISSUE CONDITIONS: (continued)	Instructions 170 through 173 in process, unit busy (VL) + 4 CPs†
	For instructions 170 and 172, S*j* reserved (except S0)
	For instructions 171 and 173, V*j* reserved as operand
EXECUTION TIME:	Instruction issue, 1 CP
	V*j* and V*k* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 11 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	(S*j*)=0 if *j*=0.

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

INSTRUCTION 174

CAL Syntax	Description	Octal Code
V*i* /HV*j*	Floating-point reciprocal approximation of (V*j* elements) to V*i* elements	174*ij*0

Instruction 174 is executed in the Reciprocal Approximation functional unit. The instruction forms an approximate value of the reciprocal of the normalized floating-point quantity in each element of V*j* and enters the result into elements of V*i*. The number of elements for which approximations are found is determined by the contents of the VL register.

Instruction 174 occurs in the divide sequence to compute the quotients of floating-point quantities as described in section 4 under floating-point arithmetic.

The reciprocal approximation instruction produces results of 30 significant bits. The low-order 18 bits are zeros. The number of significant bits can be extended to 48 using the reciprocal iteration instruction and a multiply.

HOLD ISSUE CONDITIONS:	V*i* reserved as operand or result
	V*j* reserved as operand
	Instruction 174 in process, unit busy for (VL) + 4 CPs†
EXECUTION TIME:	Instruction issue, 1 CP
	V*j* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 19 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†
SPECIAL CASES:	(V*i* element) is meaningless if (V*j* element) is not normalized; the unit assumes that bit 2^{47} of (V*j* element) is 1; no test of this bit is made.

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

CAL Syntax	Description	Octal Code
V*i* PV*j*	Population count of (V*j* elements) to V*i* elements	174*ij*1
V*i* QV*j*	Population count parity of (V*j* elements) to V*i* elements	174*ij*2

Instructions 174*ij*1 and 174*ij*2 are executed in the Vector Population/Parity functional unit, sharing some logic with the Reciprocal Approximation functional unit.

Instruction 174*ij*1 counts the number of bits set to 1 in each element of V*j* and enters the results into corresponding elements of V*i*. The results are entered into the low-order 7 bits of each V*i* element; the remaining high-order bits of each V*i* element are zeroed.

Instruction 174*ij*2 counts the number of bits set to 1 in each element of V*j*. The least significant bit of each element result shows whether the result is an odd or even number. Only the least significant bit of each element is transferred to the least significant bit position of the corresponding element of register V*i*. The remainder of the element is set to zeros. The actual population count results are not transferred.

HOLD ISSUE CONDITIONS:	V*i* reserved as operand or result
	V*j* reserved as operand
	Instructions 174*xx*1 and 174*xx*2 in process, unit busy for (VL) + 4 CPs†
	Instruction 174*xx*0 in process, unit busy for (VL) + 9 CPs†
	Instruction 070 in process, unit busy (070 issue time) + 7 CPs†

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

EXECUTION TIME:	Instruction issue, 1 CP
	V*j* ready in (VL) + 3 CPs if data available†
	V*i* ready in (VL) + 10 CPs if data available†
	Unit ready, (VL) + 4 CPs if data available†

† Vector instructions may or may not start execution immediately; they execute as data becomes available. In particular, a memory conflict that slows execution of some elements of a vector load can cause delays in all instructions in the operation chain, starting with that load.

INSTRUCTION 175

CAL Syntax	Description	Octal Code
VM V*j*,Z	VM=1 when (V*j* element)=0	1750*j*0
VM V*j*,N	VM=1 when (V*j* element)≠0	1750*j*1
VM V*j*,P	VM=1 when (V*j* element) positive, (bit 2^{63}=0), includes (V*j* element)=0	1750*j*2
VM V*j*,M	VM=1 when (V*j* element) negative, (bit 2^{63}=1)	1750*j*3
V*i*,VM V*j*,Z	VM=1 and (V*i* compress element)=element index when (V*j* element)=0	175*ij*4
V*i*,VM V*j*,N	VM=1 and (V*i* compress element)=element index when (V*j* element)≠0	175*ij*5
V*i*,VM V*j*,P	VM=1 and (V*i* compress element)=element index when (V*j* element) positive, (bit 2^{63}=0), includes (V*j* element)=0	175*ij*6
V*i*,VM V*j*,M	VM=1 and (V*i* compress element)=element index when (V*j* element) negative, (bit 2^{63}=1)	175*ij*7

Vector mask and compress index instruction 175 is executed in the Full Vector Logical functional unit.

Instruction 1750*jk*, where *k*=0 through 3, creates a vector mask in VM based on the results of testing the contents of the elements of register V*j*. Each bit of VM corresponds to an element of V*j*. Bit 2^{63} corresponds to element 0; bit 2^0 corresponds to element 63.

Instruction 175*ijk*, where *k*=4 through 7, creates an identical vector mask as in 1750*jk* and in addition creates a compressed index list in register V*i* based on the results of testing the contents of the elements of register V*j* (see example).

The type of test made by the instruction depends on the low-order 2 bits of the *k* designator. The high-order bit of the *k* designator is used to select the compress index option.

If the *k* designator is 0, the VM bit is set to 1 when (V*j* element) is 0 and is set to 0 when (V*j* element) is nonzero.

If the *k* designator is 1, the VM bit is set to 1 when (V*j* element) is nonzero and is set to 0 when (V*j* element) is 0.

If the *k* designator is 2, the VM bit is set to 1 when (V*j* element) is positive and is set to 0 when (V*j* element) is negative. A zero value is considered positive.

If the *k* designator is 3, the VM bit is set to 1 when (V*j* element) is negative and is set to 0 when (V*j* element) is positive. A zero value is considered positive.

If the *k* designator is 4, the VM bit is set to 1 and register (V*i* compress element) is set to V*j* element index when (V*j* element) is 0. Register V*i* elements are written to and V*i* element pointer advanced only when (V*j* element) is 0.

If the *k* designator is 5, the VM bit is set to 1 and register (V*i* compress element) is set to V*j* element index when (V*j* element) is nonzero. Register V*i* elements are written to and V*i* element pointer advanced only when (V*j* element) is nonzero.

If the *k* designator is 6, the VM bit is set to 1 and register (V*i* compress element) is set to V*j* element index when (V*j* element) is positive. Register V*i* elements are written to and V*i* element pointer advanced only when (V*j* element) is positive. A zero value is considered positive.

If the *k* designator is 7, the VM bit is set to 1 and register (V*i* compress element) is set to V*j* element index when (V*j* element) is negative. Register V*i* elements are written to and V*i* element pointer advanced only when (V*j* element) is negative.

The number of elements tested is determined by the contents of the VL register. VM bits corresponding to untested elements of V*j* are zeroed.

Vector mask instruction 175*jk*, *k*=0 through 3, and compress index instruction 175*ijk*, *k*=4 through 7, provide a vector counterpart to the scalar conditional branch instructions.

HOLD ISSUE CONDITIONS: V*j* reserved as operand

Instruction 14*x* in process, unit busy (VL) + 4 CPs

Instruction 175 in process, unit busy (VL) + 4 CPs

For instruction 175 (*k*=4 through 7), if register V*i* reserved as operand or result.

EXECUTION TIME:

Instruction issue, 1 CP

Vj ready, (VL) + 3 CPs if data available

For instruction 175 (k=4 through 7), Vi ready in (VL) + 10 CPs if data is available.

Except for instruction 073, VM ready (VL) + 4 CPs if data available

For instruction 073, VM ready (VL) + 5 CPs if data available

SPECIAL CASES:

k=0 or 4, VM bit xx=1 if (Vj element xx)=0.

k=1 or 5, VM bit xx=1 if (Vj element xx)$\neq$0.

k=2 or 6, VM bit xx=1 if (Vj element xx) is positive; 0 is a positive condition.

k=3 or 7, VM bit xx=1 if (Vj element xx) is negative.

k=4, (Vi compress element)=xx if (Vj element xx)=0.

k=5, (Vi compress element)=xx if (Vj element xx)$\neq$0.

k=6, (Vi compress element)=xx if (Vj element xx) is positive; 0 is a positive condition.

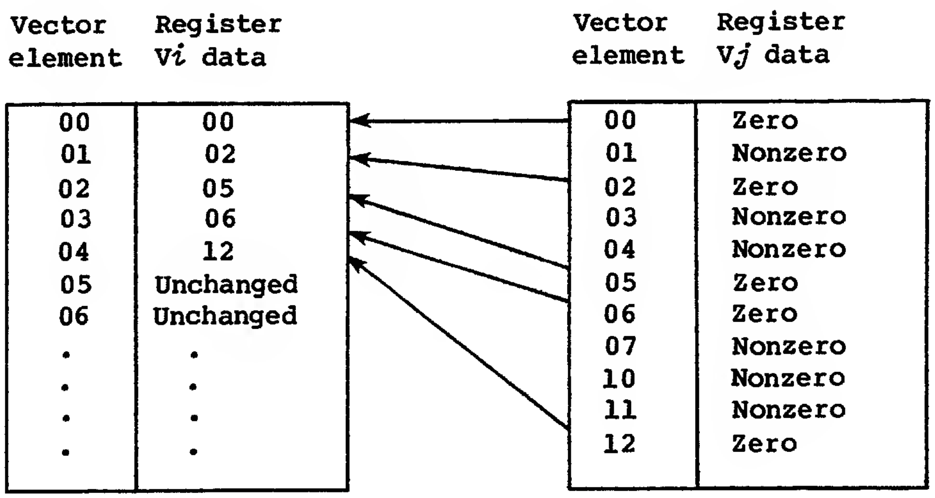
k=7, (Vi compress element)=xx if (Vj element xx) is negative.

For instruction 175 (k=4 through 7), if no test conditions are true, then (VM)=0 and no writes to register Vi occur and the elements of Vi will be unchanged by this instruction.

Example:

This example of the compress index instruction 175*ij*4 generates the same vector mask as instruction 1750*j*0 and also generates data into vector register V*i* as follows:

Vector length=13_8



Vector element	Register V*i* data
00	00
01	02
02	05
03	06
04	12
05	Unchanged
06	Unchanged
.	.
.	.
.	.
.	.

Vector element	Register V*j* data
00	Zero
01	Nonzero
02	Zero
03	Nonzero
04	Nonzero
05	Zero
06	Zero
07	Nonzero
10	Nonzero
11	Nonzero
12	Zero

CAL Syntax	Description	Octal Code
V*i* ,A0,A*k*	Transmit (VL) words from memory to V*i* elements starting at memory address (A0) and incrementing by (A*k*) for successive addresses	176*i*0*k*
V*i* ,A0,1	Transmit (VL) words from memory to V*i* elements starting at memory address (A0) and incrementing by 1 for successive addresses	176*i*00
V*i* ,A0,V*k*	Transmit (VL) words from memory to V*i* elements using memory address (A0) + (V*k* elements)	176*i*1*k*
,A0,A*k* V*j*	Transmit (VL) words from V*j* elements to memory starting at memory address (A0) and incrementing by (A*k*) for successive addresses	1770*jk*
,A0,1 V*j*	Transmit (VL) words from V*j* elements to memory starting at memory address (A0) and incrementing by 1 for successive addresses	1770*j*0
,A0,V*k* V*j*	Transmit (VL) words from V*j* elements to memory using memory address (A0) + (V*k* elements)	1771*jk*

Instructions 176 and 177 transfer blocks of data between V registers and memory.

Instruction 176 transfers data from memory to elements of register V*i*.

Instruction 177 transfers data from elements of register V*j* to memory.

For instructions 176*i*0*k* and 1770*jk*, register elements begin with 0 and are incremented by 1 for each transfer. Memory addresses begin with (A0) and are incremented by the contents of A*k*. A*k* contains a signed 24-bit integer which is added to the address of the current word to obtain the address of the next word. A*k* can specify either a positive or negative increment allowing both forward and backward streams of reference.

The number of words transferred is determined by the contents of the VL register.

For instructions 176*i*1*k* and 1771*jk*, register elements begin with 0 and are incremented by 1 for each transfer. The low-order 24 bits of each element of V*k* contains a signed 24-bit integer which is added to (A0) to obtain the current memory address.

The number of words transferred is determined by the contents of the VL register.

HOLD ISSUE CONDITIONS: For instruction 176 if Ports A and B busy

For instruction 177 if Port C busy

For instructions 176*i*1*k* and 1771*jk*, if 176*i*1*k* or 1771*jk* in progress

A0 reserved

For instructions 176*i*0*k* and 1770*jk*, if A*k* reserved where *k*=1 through 7

Scalar reference in CP1, CP2, or CP3

For instruction 176, V register *i* reserved as operand or result

For instruction 177, V register *j* reserved as operand

For instruction 176*i*1*k* and 1771*jk*, V register *k* reserved as operand

If not bidirectional memory mode, then instruction 176 holds on Port C busy and instruction 177 holds on Port A or B busy.

EXECUTION TIME: For instruction 176*i*0*k*:
- Instruction issue, 1 CP
- V*i* ready, (VL) + 17 CPs if memory is available
- Port A or B busy, (VL) + 6 CPs

For instruction 1770*jk*:
- Instruction issue, 1 CP
- V*j* ready, (VL) +3 CPs if data is available
- Port C busy, (VL) + 7 CPs

For instruction 176*i*1*k*:
- Instruction issue, 1 CP

Vi ready, (VL) + 21 CPs if memory is available
Vk ready, (VL) + 3 CPs if data is available
Port A or B busy, (VL) + 10 CPs
176*i*1*k* busy, (VL) + 10 CPs

For instruction 1771*jk*:
Instruction issue, 1 CP
Vi and *Vk* ready, (VL) + 3 CPs if data is available
Port C busy, (VL) + 10 CPs
1771*jk* busy, (VL) + 10 CPs

SPECIAL CASES:

For instructions 176*i*0*k* and 1770*jk*, increment (A0)=1 if *k*=0.

Instruction 176 uses Port B. If Port B is busy at issue time, instruction 176 uses Port A. Instruction 177 uses Port C.

For instructions 176*i*0*k* and 1770*jk*: (A*k*) determines the memory increment. Successive addresses are located in successive banks. References to the same bank can be made every 4 CPs or more. Incrementing (A*k*) by 64 places successive memory references in the same bank, so a word is transferred every 4 CPs or more. If the address is incremented by 32, every other reference is to the same bank, and words can transfer no faster than one every 2 CPs. With any address incrementing that allows 4 CPs before addressing the same bank, the words can transfer each CP.

Memory conflict can slow loading or storing of individual vector elements. The elements are loaded or stored in order, so any delay for any element delays all succeeding elements.

For instruction 176, if there is an instruction using its destination register as a source, the execution of that instruction is delayed whenever there is a delay in instruction 176 results.

APPENDIX SECTION

INSTRUCTION SUMMARY FOR CRAY X-MP MODEL 48

CRAY X-MP	CAL	UNIT	DESCRIPTION
000000	ERR	-	Error exit
††0010*jk*	CA,A*j* A*k*	-	Set the channel (A*j*) current address to (A*k*) and begin the I/O sequence
††0011*jk*	CL,A*j* A*k*	-	Set the channel (A*j*) limit address to (A*k*)
††0012*j*0	CI,A*j*	-	Clear Channel (A*j*) Interrupt flag; clear device master-clear (output channel).
††0012*j*1	MC,A*j*	-	Clear Channel (A*j*) Interrupt flag; set device master-clear (output channel); clear device ready-held (input channel).
††0013*j*0	XA A*j*	-	Enter XA register with (A*j*)
††0014*j*0	RT S*j*	-	Enter RTC register with (S*j*)
††0014*j*1	IP,*j*1	-	Set interprocessor interrupt
††001402	IP 0	-	Clear interprocessor interrupt
††001403	CLN 0	-	Enter CLN register with 0
††001413	CLN 1	-	Enter CLN register with 1
††001423	CLN 2	-	Enter CLN register with 2
††001433	CLN 3	-	Enter CLN register with 3
††001443	CLN 4	-	Enter CLN register with 4
††001453	CLN 5	-	Enter CLN register with 5
††0014*j*4	PCI S*j*	-	Enter II register with (S*j*)
††001405	CCI	-	Clear PCI request
††001406	ECI	-	Enable PCI request
††001407	DCI	-	Disable PCI request
††0015*j*0	†††	-	Select performance monitor
††001501	†††	-	Set maintenance read mode
††001511	†††	-	Load diagnostic checkbyte with S1
††001521	†††	-	Set maintenance write mode 1
††001531	†††	-	Set maintenance write mode 2
00200*k*	VL A*k*	-	Transmit (A*k*) to VL register
†002000	VL 1	-	Transmit 1 to VL register
002100	EFI	-	Enable interrupt on floating-point error

† Special syntax form
†† Privileged to monitor mode
††† Not supported at this time

CRAY X-MP	CAL	UNIT	DESCRIPTION
002200	DFI	-	Disable interrupt on floating-point error
002300	ERI	-	Enable operand range interrupts
002400	DRI	-	Disable operand range interrupts
002500	DBM	-	Disable bidirectional memory transfers
002600	EBM	-	Enable bidirectional memory transfers
002700	CMR	-	Complete memory references
0030*j*0	VM S*j*	-	Transmit (S*j*) to VM register
†003000	VM 0	-	Clear VM register
0034*jk*	SM*jk* 1,TS	-	Test & set semaphore *jk* in SM
0036*jk*	SM*jk* 0	-	Clear semaphore *jk* in SM
0037*jk*	SM*jk* 1	-	Set semaphore *jk* in SM
004000	EX	-	Normal exit
0050*jk*	J B*jk*	-	Jump to (B*jk*)
006*ijkm*	J *exp*	-	Jump to *exp*
007*ijkm*	R *exp*	-	Return jump to *exp*; set B00 to P.
010*ijkm*	JAZ *exp*	-	Branch to *exp* if (A0)=0
011*ijkm*	JAN *exp*	-	Branch to *exp* if (A0)≠0
012*ijkm*	JAP *exp*	-	Branch to *exp* if (A0) positive; 0 is positive.
013*ijkm*	JAM *exp*	-	Branch to *exp* if (A0) negative
014*ijkm*	JSZ *exp*	-	Branch to *exp* if (S0)=0
015*ijkm*	JSN *exp*	-	Branch to *exp* if (S0)≠0
016*ijkm*	JSP *exp*	-	Branch to *exp* if (S0) positive; 0 is positive.
017*ijkm*	JSM *exp*	-	Branch to *exp* if (S0) negative
01*hijkm*	A*h* *exp*	-	Transmit *exp*=*ijkm* to A*h*
020*ijkm*	A*i* *exp*	-	Transmit *exp*=*jkm* to A*i*
021*ijkm*	A*i* *exp*	-	Transmit *exp*=ones complement of *jkm* to A*i*
022*ijk*	A*i* *exp*	-	Transmit *exp*=*jk* to A*i*
023*ij*0	A*i* S*j*	-	Transmit (S*j*) to A*i*
023*i*01	A*i* VL	-	Transmit (VL) to A*i*
024*ijk*	A*i* B*jk*	-	Transmit (B*jk*) to A*i*
025*ijk*	B*jk* A*i*	-	Transmit (A*i*) to B*jk*
026*ij*0	A*i* PS*j*	Pop/LZ	Population count of (S*j*) to A*i*
026*ij*1	A*i* QS*j*	Pop/LZ	Population count parity of (S*j*) to A*i*
026*ij*7	A*i* SB*j*	-	Transmit (SB*j*) to A*i*
027*ij*0	A*i* ZS*j*	Pop/LZ	Leading zero count of (S*j*) to A*i*
027*ij*7	SB*j* A*i*	-	Transmit (A*i*) to SB*j*
030*ijk*	A*i* A*j*+A*k*	A Int Add	Integer sum of (A*j*) and (A*k*) to A*i*
†030*i*0*k*	A*i* A*k*	A Int Add	Transmit (A*k*) to A*i*
†030*ij*0	A*i* A*j*+1	A Int Add	Integer sum of (A*j*) and 1 to A*i*

† Special syntax form

CRAY X-MP	CAL	UNIT	DESCRIPTION
031*ijk*	A*i* A*j*-A*k*	A Int Add	Integer difference of (A*j*) less (A*k*) to A*i*
†031*i*00	A*i* -1	A Int Add	Transmit -1 to A*i*
†031*i*0*k*	A*i* -A*k*	A Int Add	Transmit the negative of (A*k*) to A*i*
†031*ij*0	A*i* A*j*-1	A Int Add	Integer difference of (A*j*) less 1 to A*i*
032*ijk*	A*i* A*j**A*k*	A Int Mult	Integer product of (A*j*) and (A*k*) to A*i*
033*i*00	A*i* CI	-	Channel number to A*i* (*j*=0)
033*ij*0	A*i* CA,A*j*	-	Address of channel (A*j*) to A*i* (*j*≠0; *k*=0)
033*ij*1	A*i* CE,A*j*	-	Error flag of channel (A*j*) to A*i* (*j*≠0; *k*=1)
034*ijk*	B*jk*,A*i* ,A0	Memory	Read (A*i*) words to B register *jk* from (A0)
†034*ijk*	B*jk*,A*i* 0,A0	Memory	Read (A*i*) words to B register *jk* from (A0)
035*ijk*	,A0 B*jk*,A*i*	Memory	Store (A*i*) words at B register *jk* to (A0)
†035*ijk*	0,A0 B*jk*,A*i*	Memory	Store (A*i*) words at B register *jk* to (A0)
036*ijk*	T*jk*,A*i* ,A0	Memory	Read (A*i*) words to T register *jk* from (A0)
†036*ijk*	T*jk*,A*i* 0,A0	Memory	Read (A*i*) words to T register *jk* from (A0)
037*ijk*	,A0 T*jk*,A*i*	Memory	Store (A*i*) words at T register *jk* to (A0)
†037*ijk*	0,A0 T*jk*,A*i*	Memory	Store (A*i*) words at T register *jk* to (A0)
040*ijkm*	S*i* *exp*	-	Transmit *jkm* to S*i*
041*ijkm*	S*i* *exp*	-	Transmit *exp*=ones complement of *jkm* to S*i*
042*ijk*	S*i* <*exp*	S Logical	Form ones mask *exp* bits in S*i* from the right; *jk* field gets 64-*exp*.
†042*ijk*	S*i* #>*exp*	S Logical	Form zeros mask *exp* bits in S*i* from the left; *jk* field gets 64-*exp*.
†042*i*77	S*i* 1	S Logical	Enter 1 into S*i*
†042*i*00	S*i* -1	S Logical	Enter -1 into S*i*
043*ijk*	S*i* >*exp*	S Logical	Form ones mask *exp* bits in S*i* from the left; *jk* field gets *exp*.
†043*ijk*	S*i* #<*exp*	S Logical	Form zeros mask *exp* bits in S*i* from the right; *jk* field gets 64-*exp*.
†043*i*00	S*i* 0	S Logical	Clear S*i*

† Special syntax form

CRAY X-MP	CAL	UNIT	DESCRIPTION
044*ijk*	S*i* S*j*&S*k*	S Logical	Logical product of (S*j*) and (S*k*) to S*i*
†044*ij*0	S*i* S*j*&SB	S Logical	Sign bit of (S*j*) to S*i*
†044*ij*0	S*i* SB&S*j*	S Logical	Sign bit of (S*j*) to S*i* (*j*≠0)
045*ijk*	S*i* #S*k*&S*j*	S Logical	Logical product of (S*j*) and ones complement of (S*k*) to S*i*
†045*ij*0	S*i* #SB&S*j*	S Logical	(S*j*) with sign bit cleared to S*i*
046*ijk*	S*i* S*j*\S*k*	S Logical	Logical difference of (S*j*) and (S*k*) to S*i*
†046*ij*0	S*i* S*j*\SB	S Logical	Toggle sign bit of S*j*, then enter into S*i*
†046*ij*0	S*i* SB\S*j*	S Logical	Toggle sign bit of S*j*, then enter into S*i* (*j*≠0)
047*ijk*	S*i* #S*j*\S*k*	S Logical	Logical equivalence of (S*k*) and (S*j*) to S*i*
†047*i*0*k*	S*i* #S*k*	S Logical	Transmit ones complement of (S*k*) to S*i*
†047*ij*0	S*i* #S*j*\SB	S Logical	Logical equivalence of (S*j*) and sign bit to S*i*
†047*ij*0	S*i* #SB\S*j*	S Logical	Logical equivalence of (S*j*) and sign bit to S*i* (*j*≠0)
†047*i*00	S*i* #SB	S Logical	Enter ones complement of sign bit into S*i*
050*ijk*	S*i* S*j*!S*i*&S*k*	S Logical	Logical product of (S*i*) and (S*k*) complement ORed with logical product of (S*j*) and (S*k*) to S*i*
†050*ij*0	S*i* S*j*!S*i*&SB	S Logical	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*
051*ijk*	S*i* S*j*!S*k*	S Logical	Logical sum of (S*j*) and (S*k*) to S*i*
†051*i*0*k*	S*i* S*k*	S Logical	Transmit (S*k*) to S*i*
†051*ij*0	S*i* S*j*!SB	S Logical	Logical sum of (S*j*) and sign bit to S*i*
†051*ij*0	S*i* SB!S*j*	S Logical	Logical sum of (S*j*) and sign bit to S*i* (*j*≠0)
†051*i*00	S*i* SB	S Logical	Enter sign bit into S*i*
052*ijk*	S0 S*i*<*exp*	S Shift	Shift (S*i*) left *exp*=*jk* places to S0
053*ijk*	S0 S*i*>*exp*	S Shift	Shift (S*i*) right *exp*=64-*jk* places to S0
054*ijk*	S*i* S*i*<*exp*	S Shift	Shift (S*i*) left *exp*=*jk* places
055*ijk*	S*i* S*i*>*exp*	S Shift	Shift (S*i*) right *exp*=64-*jk* places
056*ijk*	S*i* S*i*,S*j*<A*k*	S Shift	Shift (S*i* and S*j*) left (A*k*) places to S*i*

† Special syntax form

CRAY X-MP	CAL	UNIT	DESCRIPTION
†056*ij*0	S*i* S*i*,S*j*<1	S Shift	Shift (S*i* and S*j*) left one place to S*i*
†056*i*0*k*	S*i* S*i*<A*k*	S Shift	Shift (S*i*) left (A*k*) places to S*i*
057*ijk*	S*i* S*j*,S*i*>A*k*	S Shift	Shift (S*j* and S*i*) right (A*k*) places to S*i*
†057*ij*0	S*i* S*j*,S*i*>1	S Shift	Shift (S*j* and S*i*) right one place to S*i*
†057*i*0*k*	S*i* S*i*>A*k*	S Shift	Shift (S*i*) right (A*k*) places to S*i*
060*ijk*	S*i* S*j*+S*k*	S Int Add	Integer sum of (S*j*) and (S*k*) to S*i*
061*ijk*	S*i* S*j*-S*k*	S Int Add	Integer difference of (S*j*) and (S*k*) to S*i*
†061*i*0*k*	S*i* -S*k*	S Int Add	Transmit negative of (S*k*) to S*i*
062*ijk*	S*i* S*j*+FS*k*	Fp Add	Floating-point sum of (S*j*) and (S*k*) to S*i*
†062*i*0*k*	S*i* +FS*k*	Fp Add	Normalize (S*k*) to S*i*
063*ijk*	S*i* S*j*-FS*k*	Fp Add	Floating-point difference of (S*j*) and (S*k*) to S*i*
†063*i*0*k*	S*i* -FS*k*	Fp Add	Transmit normalized negative of (S*k*) to S*i*
064*ijk*	S*i* S*j**FS*k*	Fp Mult	Floating-point product of (S*j*) and (S*k*) to S*i*
065*ijk*	S*i* S*j**HS*k*	Fp Mult	Half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
066*ijk*	S*i* S*j**RS*k*	Fp Mult	Full-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
067*ijk*	S*i* S*j**IS*k*	Fp Mult	2-floating-point product of (S*j*) and (S*k*) to S*i*
070*ij*0	S*i* /HS*j*	Fp Rcpl	Floating-point reciprocal approximation of (S*j*) to S*i*
071*i*0*k*	S*i* A*k*	-	Transmit (A*k*) to S*i* with no sign extension
071*i*1*k*	S*i* +A*k*	-	Transmit (A*k*) to S*i* with sign extension
071*i*2*k*	S*i* +FA*k*	-	Transmit (A*k*) to S*i* as unnormalized floating-point number
071*i*30	S*i* 0.6	-	Transmit constant 0.75*2**48 to S*i*
071*i*40	S*i* 0.4	-	Transmit constant 0.5 to S*i*
071*i*50	S*i* 1.	-	Transmit constant 1.0 to S*i*
071*i*60	S*i* 2.	-	Transmit constant 2.0 to S*i*

† Special syntax form

CRAY X-MP	CAL	UNIT	DESCRIPTION
071*i*70	S*i* 4.	-	Transmit constant 4.0 to S*i*
072*i*00	S*i* RT	-	Transmit (RTC) to S*i*
072*i*02	S*i* SM	-	Transmit (SM) to S*i*
072*ij*3	S*i* ST*j*	-	Transmit (ST*j*) to S*i*
073*i*00	S*i* VM	-	Transmit (VM) to S*i*
073*i*11	††	-	Read counter into S*i*
073*i*21	††	-	Increment performance counter (maintenance)
073*i*31	††	-	Clear all maintenance modes
073*ij*1	S*i* SR*j*	-	Transmit (SR*j*) to S*i* (*j*=0)
073*i*02	SM S*i*	-	Transmit (S*i*) to SM
073*ij*3	ST*j* S*i*	-	Transmit (S*i*) to ST*j*
074*ijk*	S*i* T*jk*	-	Transmit (T*jk*) to S*i*
075*ijk*	T*jk* S*i*	-	Transmit (S*i*) to T*jk*
076*ijk*	S*i* V*j*,A*k*	-	Transmit (V*j*, element (A*k*)) to S*i*
077*ijk*	V*i*,A*k* S*j*	-	Transmit (S*j*) to V*i* element (A*k*)
†077*i*0*k*	V*i*,A*k* 0	-	Clear V*i* element (A*k*)
10h*ijkm*	A*i* *exp*,A*h*	Memory	Read from ((A*h*)+*exp*) to A*i* (A0=0)
†100*ijkm*	A*i* *exp*,0	Memory	Read from (*exp*) to A*i*
†100*ijkm*	A*i* *exp*,	Memory	Read from (*exp*) to A*i*
†10*hi*00 0	A*i* ,A*h*	Memory	Read from (A*h*) to A*i*
11*hijkm*	*exp*,A*h* A*i*	Memory	Store (A*i*) to (A*h*)+*exp* (A0=0)
†110*ijkm*	*exp*,0 A*i*	Memory	Store (A*i*) to *exp*
†110*ijkm*	*exp*, A*i*	Memory	Store (A*i*) to *exp*
†11*hi*00 0	,A*h* A*i*	Memory	Store (A*i*) to (A*h*)
12*hijkm*	S*i* *exp*,A*h*	Memory	Read from ((A*h*)+*exp*) to S*i* (A0=0)
†120*ijkm*	S*i* *exp*,0	Memory	Read from *exp* to S*i*
†120*ijkm*	S*i* *exp*,	Memory	Read from *exp* to S*i*
†12*hi*00 0	S*i* ,A*h*	Memory	Read from (A*h*) to S*i*
13*hijkm*	*exp*,A*h* S*i*	Memory	Store (S*i*) to (A*h*)+*exp* (A0=0)
†130*ijkm*	*exp*,0 S*i*	Memory	Store (S*i*) to *exp*
†130*ijkm*	*exp*, S*i*	Memory	Store (S*i*) to *exp*
†13*hi*00 0	,A*h* S*i*	Memory	Store (S*i*) to (A*h*)
140*ijk*	V*i* S*j*&V*k*	V Logical	Logical products of (S*j*) and (V*k*) to V*i*
141*ijk*	V*i* V*j*&V*k*	V Logical	Logical products of (V*j*) and (V*k*) to V*i*
142*ijk*	V*i* S*j*!V*k*	V Logical	Logical sums of (S*j*) and (V*k*) to V*i*
†142*i*0*k*	V*i* V*k*	V Logical	Transmit (V*k*) to V*i*
143*ijk*	V*i* V*j*!V*k*	V Logical	Logical sums of (V*j*) and (V*k*) to V*i*

† Special syntax form
†† Not supported at this time

CRAY X-MP	CAL		UNIT	DESCRIPTION
144ijk	Vi	Sj\Vk	V Logical	Logical differences of (Sj) and (Vk) to Vi
145ijk	Vi	Vj\Vk	V Logical	Logical differences of (Vj) and (Vk) to Vi
†145iii	Vi	0	V Logical	Clear Vi
146ijk	Vi	Sj!Vk&VM	V Logical	Transmit (Sj) if VM bit=1; (Vk) if VM bit=0 to Vi.
†146i0k	Vi	#VM&Vk	V Logical	Vector merge of (Vk) and 0 to Vi
147ijk	Vi	Vj!Vk&VM	V Logical	Transmit (Vj) if VM bit=1; (Vk) if VM bit=0 to Vi.
150ijk	Vi	Vj<Ak	V Shift	Shift (Vj) left (Ak) places to Vi
†150ij0	Vi	Vj<1	V Shift	Shift (Vj) left one place to Vi
151ijk	Vi	Vj>Ak	V Shift	Shift (Vj) right (Ak) places to Vi
†151ij0	Vi	Vj>1	V Shift	Shift (Vj) right one place to Vi
152ijk	Vi	Vj,Vj<Ak	V Shift	Double shift (Vj) left (Ak) places to Vi
†152ij0	Vi	Vj,Vj<1	V Shift	Double shift (Vj) left one place to Vi
153ijk	Vi	Vj,Vj>Ak	V Shift	Double shift (Vj) right (Ak) places to Vi
†153ij0	Vi	Vj,Vj>1	V Shift	Double Shift (Vj) right one place to Vi
154ijk	Vi	Sj+Vk	V Int Add	Integer sums of (Sj) and (Vk) to Vi
155ijk	Vi	Vj+Vk	V Int Add	Integer sums of (Vj) and (Vk) to Vi
156ijk	Vi	Sj-Vk	V Int Add	Integer differences of (Sj) and (Vk) to Vi
†156i0k	Vi	-Vk	V Int Add	Transmit negative of (Vk) to Vi
157ijk	Vi	Vj-Vk	V Int Add	Integer differences of (Vj) and (Vk) to Vi
160ijk	Vi	Sj*FVk	Fp Mult	Floating-point products of (Sj) and (Vk) to Vi
161ijk	Vi	Vj*FVk	Fp Mult	Floating-point products of (Vj) and (Vk) to Vi
162ijk	Vi	Sj*HVk	Fp Mult	Half-precision rounded floating-point products of (Sj) and (Vk) to Vi
163ijk	Vi	Vj*HVk	Fp Mult	Half-precision rounded floating-point products of (Vj) and (Vk) to Vi
164ijk	Vi	Sj*RVk	Fp Mult	Rounded floating-point products of (Sj) and (Vk) to Vi

† Special syntax form

CRAY X-MP	CAL		UNIT	DESCRIPTION
165*ijk*	V*i*	V*j**RV*k*	Fp Mult	Rounded floating-point products of (V*j*) and (V*k*) to V*i*
166*ijk*	V*i*	S*j**IV*k*	Fp Mult	2-floating-point products of (S*j*) and (V*k*) to V*i*
167*ijk*	V*i*	V*j**IV*k*	Fp Mult	2-floating-point products of (V*j*) and (V*k*) to V*i*
170*ijk*	V*i*	S*j*+FV*k*	Fp Add	Floating-point sums of (S*j*) and (V*k*) to V*i*
†170*i*0*k*	V*i*	+FV*k*	Fp Add	Normalize (V*k*) to V*i*
171*ijk*	V*i*	V*j*+FV*k*	Fp Add	Floating-point sums of (V*j*) and (V*k*) to V*i*
172*ijk*	V*i*	S*j*-FV*k*	Fp Add	Floating-point differences of (S*j*) and (V*k*) to V*i*
†172*i*0*k*	V*i*	-FV*k*	Fp Add	Transmit normalized negatives of (V*k*) to V*i*
173*ijk*	V*i*	V*j*-FV*k*	Fp Add	Floating-point differences of (V*j*) and (V*k*) to V*i*
174*ij*0	V*i*	/HV*j*	Fp Rcpl	Floating-point reciprocal approximations of (V*j*) to V*i*
174*ij*1	V*i*	PV*j*	V Pop	Population counts of (V*j*) to V*i*
174*ij*2	V*i*	QV*j*	V Pop	Population count parities of (V*j*) to V*i*
1750*j*0	VM	V*j*,Z	V Logical	VM=1 where (V*j*)=0
1750*j*1	VM	V*j*,N	V Logical	VM=1 where (V*j*)≠0
1750*j*2	VM	V*j*,P	V Logical	VM=1 if (V*j*) positive; 0 is positive.
1750*j*3	VM	V*j*,M	V Logical	VM=1 if (V*j*) negative
1750*j*4	V*i*,VM	V*j*,Z	V Logical	VM=1 and (V*i*)=element index if (V*j*)=0
1750*j*5	V*i*,VM	V*j*,N	V Logical	VM=1 and (V*i*)=element index if (V*j*)≠0
1750*j*6	V*i*,VM	V*j*,P	V Logical	VM=1 and (V*i*)=element index if (V*j*) positive
1750*j*7	V*i*,VM	V*j*,M	V Logical	VM=1 and (V*i*)=element index if (V*j*) negative
176*i*0*k*	V*i*	,A0,A*k*	Memory	Read (VL) words to V*i* from (A0) incremented by (A*k*)
†176*i*00	V*i*	,A0,1	Memory	Read (VL) words to V*i* from (A0) incremented by 1
176*i*1*k*	V*i*	,A0,V*k*	Memory	Read (VL) words to V*i* using (A0) + (V*k*)
1770*jk*	,A0,A*k*	V*j*	Memory	Store (VL) words from V*j* to (A0) incremented by (A*k*)
†1770*j*0	,A0,1	V*j*	Memory	Store (VL) words from V*j* to (A0) incremented by 1
1771*jk*	,A0,V*k*	V*j*	Memory	Store (VL) words from V*j* using (A0) + (V*k*)

† Special syntax form

6 MBYTE PER SECOND CHANNEL DESCRIPTION B

INTRODUCTION

Each input or output 6 Mbyte per second channel directly accesses Central Memory. Input channels store external data in memory and output channels read data from memory. A primary task of a channel is to convert 64-bit Central Memory words into 16-bit parcels or 16-bit parcels into 64-bit Central Memory words. Four parcels make up one Central Memory word with bits of the parcels assigned to memory bit positions (see section 2 of this publication).

Each input or output channel has a data channel (4 parity bits, 16 data bits, and 3 control lines), a 64-bit assembly or disassembly register, a channel Current Address (CA) register, and a channel Limit Address (CL) register.

Three control signals (Ready, Resume, and Disconnect) coordinate the transfer of parcels over the channels. In addition to the three control signals, the output channel of the pair has a Master Clear line.

This appendix describes the signal sequence of a 6 Mbyte per second input channel and an output channel.

6 MBYTE PER SECOND INPUT CHANNEL SIGNAL SEQUENCE

A general view of a 6 Mbyte per second input channel signal sequence is illustrated in table B-1. The data bits, parity bits, and each signal in the sequence are described below.

DATA BITS 2^0 THROUGH 2^{15}

Data bits 2^0, 2^1, ..., 2^{15} are signals carrying the 16-bit parcel of data from the external device to Central Memory. The data bits must all be valid within 25 nanoseconds after the leading edge of the Ready signal. Data bit signals must remain unchanged on the lines until the corresponding Resume signal is received by the external device. Normally, data is sent coincidentally with the Ready signal and is held until the subsequent Ready signal.

Table B-1. Input channel signal exchange

Central Memory	Channel	External Equipment
1. Activate channel (set CL and CA).		
2. †	$\longleftarrow$	Data 2^{63} - 2^{48} with Ready
3. Resume	$\longrightarrow$	
4.	$\longleftarrow$	Data 2^{47} - 2^{32} with Ready
5. Resume	$\longrightarrow$	
6.	$\longleftarrow$	Data 2^{31} - 2^{16} with Ready
7. Resume	$\longrightarrow$	
8.	$\longleftarrow$	Data 2^{15} - 2^{0} with Ready
9. Write word to memory and advance current address.		
10a. Resume	$\longrightarrow$	
10b. If (CA)=(CL), go to 13.		
11.		If more data, go to 2.
12.	$\longleftarrow$	Disconnect (ignored if CA=CL or if channel not active).
13. Set interrupt and deactivate channel.		

† Step 2 can initially precede step 1; that is, the first parcel and ready signal can arrive before requested.

PARITY BITS 0 THROUGH 3

Parity bits 0, 1, 2, and 3 are each assigned to a 4-bit group of data bits. The parity bits are set or cleared to give the bit group odd parity. Bit assignments follow.

Parity bit	Data bits
0	$2^0 - 2^3$
1	$2^4 - 2^7$
2	$2^8 - 2^{11}$
3	$2^{12} - 2^{15}$

Parity bits are sent from the external device to Central Memory at the same time as data bits and are held stable in the same way as the data bits.

READY SIGNAL

The Ready signal sent to Central Memory indicates a parcel of data is being sent to the Central Memory input channel and can be sampled. A Ready signal is a pulse 50 ± 10 nanoseconds wide (at 50% voltage points). The leading edge of the Ready signal at Central Memory begins the timing for sampling the data bits.

RESUME SIGNAL

The Resume signal is sent from Central Memory to the external device showing the parcel was received and Central Memory is ready for the next data transmission. A Resume signal is a pulse 50 ± 8 nanoseconds wide (at 50% voltage points).

DISCONNECT SIGNAL

The Disconnect signal is sent from the external device to Central Memory and indicates transmission from the external device is complete. The Disconnect signal is sent after the Resume signal is received for the last Ready signal. A Disconnect signal is a pulse 50 ± 10 nanoseconds wide (at the 50% voltage points).

<u>6 MBYTE PER SECOND OUTPUT CHANNEL SIGNAL SEQUENCE</u>

A general view of a 6 Mbyte per second output channel signal sequence is illustrated in table B-2. The data bits, parity bits, and each signal in the sequence are described following the table.

Table B-2. Output channel signal exchange

Central Memory	Channel	External Equipment
1. Activate channel (set CL and CA).		
2. Read word from memory and advance current address.		
3. Data 2^{63} - 2^{48} with Ready	$\longrightarrow$	
4.	$\longleftarrow$	Resume
5. Data 2^{47} - 2^{32} with Ready	$\longrightarrow$	
6.	$\longleftarrow$	Resume
7. Data 2^{31} - 2^{16} with Ready	$\longrightarrow$	
8.	$\longleftarrow$	Resume
9. Data 2^{15} - 2^{0} with Ready	$\longrightarrow$	
10.	$\longleftarrow$	Resume
11. If $(CA)\neq(CL)$, go to 2.		
12. Disconnect.	$\longrightarrow$	
13. Set interrupt and deactivate channel.		

DATA BITS 2^0 THROUGH 2^{15}

Data bits 2^0, 2^1, ..., 2^{15} are signals carrying a 16-bit parcel of data from Central Memory to an external device. The data bits are sent concurrently within 5 nanoseconds of the leading edge of the Ready signal. Data bit signals remain steady on the lines until the Resume signal is received.

PARITY BITS 0 THROUGH 3

Parity bits 0, 1, 2, and 3 are each assigned to a 4-bit group of data bits. The parity bits are set or cleared to give the bit group odd parity. Bit assignments follow:

Parity bit	Data bits
0	$2^0 - 2^3$
1	$2^4 - 2^7$
2	$2^8 - 2^{11}$
3	$2^{12} - 2^{15}$

Parity bits are sent from Central Memory to the external device at the same time as the data bits and are held stable in the same way as the data bits.

READY SIGNAL

The Ready signal sent from Central Memory to the external device indicates data is present and can be sampled. A Ready signal is a pulse 50 ± 8 nanoseconds wide (at 50% voltage points). The leading edge of the Ready signal can be used to time data sampling in the external device.

RESUME SIGNAL

The Resume signal is sent from the external device to Central Memory showing the parcel was received and the external device is ready for the next parcel transmission. A Resume signal is a pulse 50 ± 10 nanoseconds wide (at 50% voltage points).

DISCONNECT SIGNAL

The Disconnect signal is sent from Central Memory to the external device and indicates transmission from Central Memory is complete. The Disconnect signal is sent after Central Memory receives the Resume signal from the last Ready signal. A Disconnect signal is a pulse 50 ± 8 nanoseconds wide (at 50% voltage points).

PERFORMANCE MONITOR C

INTRODUCTION

The system contains a set of eight performance counters to track certain hardware related events that can be used to indicate relative performance. The events that can be tracked are the number of specific instructions issued, hold issue conditions, the number of fetches, references, etc. and are selected through instruction 0015*j*0. Table C-1 lists all operations that can be monitored.

Performance monitoring instructions allow you to select specific hardware related events for monitoring, read the results of the performance monitors into a scalar register, and test the operation of the performance counters.

The instructions used for performance monitoring are:

0015*j*0	Select performance monitor.
073*i*11	Read performance counter into S*i*.
073*i*21	Increment performance counter (maintenance).

All instructions are executed in monitor mode.

SELECTING PERFORMANCE EVENTS

Instruction 0015*j*0 selects for monitoring one of the four groups of hardware related events shown in table C-1 and clears all performance monitors. The low-order 2 bits of the *j* field selects the group.

During each CP in non-monitor (user) mode, the performance counters advance their totals according to the number of monitored events that occur. Each of the performance counters can increment at a maximum rate of +3 per CP. This allows a counter to continuously monitor for approximately 62 hours before it is reset.

Performance events are monitored only while operating in user (non-monitor) mode. Entering monitor mode disables advancing of the performance counters.

Table C-1. Performance counter group descriptions

Monitor Function	Performance Counter	Description	Increment Per CP
		Number of:	
j=0	0	Instructions issued	+1
	1	CPs holding issue	+1
	2	Fetches	+1
	3	I/O references	+1
	4	CPU references	+3 max
	5	Floating-point add operations	+1
	6	Floating-point multiply operations	+1
	7	Floating-point reciprocal operations	+1
		Hold issue conditions:	
j=1	0	Semaphores	+1
	1	Shared registers	+1
	2	A registers and functionals	+1
	3	S registers and functionals	+1
	4	V registers	+1
	5	V functional units	+1
	6	Scalar memory	+1
	7	Block memory	+1
		Number of:	
j=2	0	Fetches	+1
	1	Scalar references	+1
	2	Scalar conflicts	+1
	3	I/O references	+1
	4	I/O conflicts	+1
	5	Block references	+3 max
	6	Block conflicts	+3 max
	7	Vector memory references	+3 max
		Number of:	
j=3	0	000 - 017 instuctions	+1
	1	020 - 137 instructions	+1
	2	140 - 157, 175 instructions	+1
	3	160 - 174 instructions	+1
	4	176, 177 instructions	+1
	5	Vector integer operations	+3 max
	6	Vector floating-point operations	+3 max
	7	Vector memory references	+3 max

READING PERFORMANCE RESULTS

Performance counter totals can be read using instruction 073*i*11, which transmits either the high-order or low-order bits of a performance counter to the high-order bits of scalar register S*i* according to the contents of the performance counter pointer.

Entering monitor mode disables advancing of all performance counters and clears the performance counter pointer. The first execution of a 073*i*11 instruction reads the low-order bits of counter 0 into S*i* and increments the performance counter pointer. The second 073*i*11 instruction reads the high-order bits of counter 0 into S*i* and again increments the pointer. After each 073*i*11 instruction, the performance counter pointer is advanced by 1. Even values of the pointer select the low-order bits of a performance counter to be read into S*i*; odd values of the pointer select the high-order bits of the performance counter to be read.

Low-order bits 0 through 25 of the performance counter are read into bits 32 through 57 of S*i*. High-order bits 26 through 45 of the performance counter are read into bits 38 through 57 of S*i*.

A sequence for reading a set of performance counters appears as follows (there must be a 2 CP delay between sequential 073*i*11 instructions):

```
073i11        Low-order bits of counter 0 to Si
2 CP delay
073i11        High-order bits of counter 1 to Si
2 CP delay
073i11        Low-order bits of counter 1 to Si
2 CP delay
073i11        High-order bits of counter 2 to Si
2 CP delay
   .                    .
   .                    .
   .                    .
```

TESTING PERFORMANCE COUNTERS

Instruction 073*i*21 is used to test the operation of the performance counters by incrementing the value stored in the counter while in monitor mode.

Entering monitor mode disables advancing of all performance counters by user programs and clears the performance counter pointer. This pointer determines which performance counter, and which bits in that counter, will be incremented. Even values of the pointer increment bits 0 and 6 of the performance counter when instruction 073*i*21 is executed, odd values of the pointer increment bit 26. The pointer is advanced from even to odd and to the next counter through instruction 073*i*11.

There must be a 1 CP delay between sequential 073*i*21 instructions.

Execution of instruction 073*i*21 loads register S*i* with all ones as a side effect of the basic 073 instruction.

SECDED MAINTENANCE FUNCTIONS D

INTRODUCTION

Modules involved with generating and interpreting the 8-bit check byte used for SECDED include logic that can be used for verifying check bit storage, check bit generation, and error detection and correction.

The instructions used for these maintenance mode functions are:

001501	Set maintenance read mode.
001511	Load diagnostic check byte with S1.
001521	Set maintenance write mode 1.
001531	Set maintenance write mode 2.
073*i*31	Clear all maintenance modes.

These instructions are all executed in monitor mode, and for instructions 0015*xx*, the maintenance mode switch (located on the mainframe's control panel) must be on or the instructions become no-ops.

VERIFICATION OF CHECK BIT STORAGE

To verify the storage ability of the SECDED check bits without moving memory modules, two instructions are used: 001501 and 001521.

The maintenance write mode 1 instruction, 001521, replaces the 8 check bits generated by the SECDED circuitry with specific bits of a data word as it is written into memory. The maintenance read mode instruction, 001501, complements the write instruction by replacing the same bits of a data word with the 8 check bits as it is read from memory.

By using the instructions together (and with error correction disabled through the switch on the mainframe's control panel), specified bits of a data word are stored and read back through the check bit storage paths and verification of SECDED check bit storage operation is accomplished.

Instruction 001521, maintenance write mode 1, and 001501, maintenance read mode, replace data bits with check bits and vice versa as shown below.

Data bit		Check bit
46		0
47		1
62		2
63	Read ⟶	3
14	⟵ Write	4
15		5
30		6
31		7

VERIFICATION OF CHECK BIT GENERATION

The maintenance read mode instruction, 001501, is used to verify the correct generation of SECDED check bits for a word of data.

When the instruction is executed, the 8 check bits for SECDED replace specific data bits as the word is read into memory (as shown above). A test program can easily extract these check bits and verify their correctness, thus checking the accuracy of the SECDED check bit circuitry.

Since the CPU replaces the data bits with check bits on all reads to memory until instruction 073*i*31 is executed (including fetch, scalar and vector reads, and I/O for the CPU), the test program should initially rewrite all of memory using the 001501 instruction to set up the SECDED check bits for a subsequent read by fetch or I/O.

Error correction must be disabled during this test.

VERIFICATION OF ERROR DETECTION AND CORRECTION

The maintenance write mode 2 instruction, 001531, and the load diagnostic check byte with S1 instruction, 001511, are used to verify operation of the SECDED circuitry.

To verify operation, a diagnostic check byte is initially loaded with the upper-order bits of register S1 through instruction 001511 as shown below:

S1 bit	Diagnostic check bit
56	0
57	1
58	2
59	3
60	4
61	5
62	6
63	7

This diagnostic check byte is then written into memory in place of the normal SECDED check bits on any subsequent CPU write to memory (writes from I/O through this CPU are not affected). With error correction enabeled (through the switch on the mainframe's control panel), a subsequent read of the memory location allows different paths within the error detection and correction circuitry to be checked out.

The diagnostic check byte retains its value until a new one is entered.

CLEARING MAINTENANCE MODE FUNCTIONS

Instruction 073*i*31, clear all maintenance modes, clears the following maintenance mode instructions:

001501	Set maintenance read mode.
001521	Set maintenance write mode 1.
001531	Set maintenance write mode 2.

A Master Clear also clears the instructions.

As a side effect of the 073*i*31 instruction, S*i* is loaded with all ones.

INDEX

INDEX

READERS COMMENT FORM

CRAY X-MP Series Model 48 Mainframe Reference Manual HR-0097

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME ____________________

JOB TITLE ____________________

FIRM ____________________

ADDRESS ____________________

CITY ____________ STATE ________ ZIP ________

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

FOLD

STAPLE

READERS COMMENT FORM

CRAY X-MP Series Model 48 Mainframe Reference Manual HR-0097

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME	
JOB TITLE	
FIRM	
ADDRESS	
CITY	STATE ZIP

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

FOLD

STAPLE

Cray Research, Inc.
Publications Department
1440 Northland Drive
Mendota Heights, MN 55120
612-452-6650
TLX 298444